How to Contact MathWorks

Latest news: www.mathworks.com
Sales and services: www.mathworks.com/sales_and_services
User community: www.mathworks.com/matlabcentral
Technical support: www.mathworks.com/support/contact_us

Phone: 508-647-7000

The MathWorks, Inc.
1 Apple Hill Drive
Natick, MA 01760-2098

Simulink® Reference


The software described in this document is furnished under a license agreement. The software may be used or copied only under the terms of the license agreement. No part of this manual may be photocopied or reproduced in any form without prior written consent from The MathWorks, Inc.

FEDERAL ACQUISITION: This provision applies to all acquisitions of the Program and Documentation by, for, or through the federal government of the United States. By accepting delivery of the Program or Documentation, the government hereby agrees that this software or documentation qualifies as commercial computer software or commercial computer software documentation as such terms are used or defined in FAR 12.212, DFARS Part 227.72, and DFARS 252.227-7014. Accordingly, the terms and conditions of this Agreement and only those rights specified in this Agreement, shall pertain to and govern the use, modification, reproduction, release, performance, display, and disclosure of the Program and Documentation by the federal government (or other entity acquiring for or through the federal government) and shall supersede any conflicting contractual terms or conditions. If this License fails to meet the government’s needs or is inconsistent in any respect with federal procurement law, the government agrees to return the Program and Documentation, unused, to The MathWorks, Inc.

Trademarks

MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See www.mathworks.com/trademarks for a list of additional trademarks. Other product or brand names may be trademarks or registered trademarks of their respective holders.

Patents

MathWorks products are protected by one or more U.S. patents. Please see www.mathworks.com/patents for more information.
<table>
<thead>
<tr>
<th>Date</th>
<th>Status</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>July 2002</td>
<td>Online only</td>
<td>Revised for Simulink 5 (Release 13)</td>
</tr>
<tr>
<td>April 2003</td>
<td>Online only</td>
<td>Revised for Simulink 5.1 (Release 13SP1)</td>
</tr>
<tr>
<td>April 2004</td>
<td>Online only</td>
<td>Revised for Simulink 5.1.1 (Release 13SP1+)</td>
</tr>
<tr>
<td>June 2004</td>
<td>Online only</td>
<td>Revised for Simulink 6 (Release 14)</td>
</tr>
<tr>
<td>October 2004</td>
<td>Online only</td>
<td>Revised for Simulink 6.1 (Release 14SP1)</td>
</tr>
<tr>
<td>March 2005</td>
<td>Online only</td>
<td>Revised for Simulink 6.2 (Release 14SP2)</td>
</tr>
<tr>
<td>September 2005</td>
<td>Online only</td>
<td>Revised for Simulink 6.3 (Release 14SP3)</td>
</tr>
<tr>
<td>March 2006</td>
<td>Online only</td>
<td>Revised for Simulink 6.4 (Release 2006a)</td>
</tr>
<tr>
<td>September 2006</td>
<td>Online only</td>
<td>Revised for Simulink 6.5 (Release 2006b)</td>
</tr>
<tr>
<td>March 2007</td>
<td>Online only</td>
<td>Revised for Simulink 6.6 (Release 2007a)</td>
</tr>
<tr>
<td>September 2007</td>
<td>Online only</td>
<td>Revised for Simulink 7.0 (Release 2007b)</td>
</tr>
<tr>
<td>March 2008</td>
<td>Online only</td>
<td>Revised for Simulink 7.1 (Release 2008a)</td>
</tr>
<tr>
<td>October 2008</td>
<td>Online only</td>
<td>Revised for Simulink 7.2 (Release 2008b)</td>
</tr>
<tr>
<td>March 2009</td>
<td>Online only</td>
<td>Revised for Simulink 7.3 (Release 2009a)</td>
</tr>
<tr>
<td>September 2009</td>
<td>Online only</td>
<td>Revised for Simulink 7.4 (Release 2009b)</td>
</tr>
<tr>
<td>March 2010</td>
<td>Online only</td>
<td>Revised for Simulink 7.5 (Release 2010a)</td>
</tr>
<tr>
<td>September 2010</td>
<td>Online only</td>
<td>Revised for Simulink 7.6 (Release 2010b)</td>
</tr>
<tr>
<td>April 2011</td>
<td>Online only</td>
<td>Revised for Simulink 7.7 (Release 2011a)</td>
</tr>
<tr>
<td>September 2011</td>
<td>Online only</td>
<td>Revised for Simulink 7.8 (Release 2011b)</td>
</tr>
<tr>
<td>March 2012</td>
<td>Online only</td>
<td>Revised for Simulink 7.9 (Release 2012a)</td>
</tr>
<tr>
<td>September 2012</td>
<td>Online only</td>
<td>Revised for Simulink 8.0 (Release 2012b)</td>
</tr>
<tr>
<td>March 2013</td>
<td>Online only</td>
<td>Revised for Simulink 8.1 (Release 2013a)</td>
</tr>
<tr>
<td>September 2013</td>
<td>Online only</td>
<td>Revised for Simulink 8.2 (Release 2013b)</td>
</tr>
<tr>
<td>March 2014</td>
<td>Online only</td>
<td>Revised for Simulink 8.3 (Release 2014a)</td>
</tr>
<tr>
<td>October 2014</td>
<td>Online only</td>
<td>Revised for Simulink 8.4 (Release 2014b)</td>
</tr>
<tr>
<td>March 2015</td>
<td>Online only</td>
<td>Revised for Simulink 8.5 (Release 2015a)</td>
</tr>
<tr>
<td>September 2015</td>
<td>Online only</td>
<td>Revised for Simulink 8.6 (Release 2015b)</td>
</tr>
<tr>
<td>October 2015</td>
<td>Online only</td>
<td>Rereleased for Simulink 8.5.1 (Release 2015aSP1)</td>
</tr>
<tr>
<td>March 2016</td>
<td>Online only</td>
<td>Revised for Simulink 8.7 (Release 2016a)</td>
</tr>
<tr>
<td>September 2016</td>
<td>Online only</td>
<td>Revised for Simulink 8.8 (Release 2016b)</td>
</tr>
<tr>
<td>March 2017</td>
<td>Online only</td>
<td>Revised for Simulink 8.9 (Release 2017a)</td>
</tr>
<tr>
<td>September 2017</td>
<td>Online only</td>
<td>Revised for Simulink 9.0 (Release 2017b)</td>
</tr>
<tr>
<td>March 2018</td>
<td>Online only</td>
<td>Revised for Simulink 9.1 (Release 2018a)</td>
</tr>
<tr>
<td>September 2018</td>
<td>Online only</td>
<td>Revised for Simulink 9.2 (Release 2018b)</td>
</tr>
<tr>
<td>March 2019</td>
<td>Online only</td>
<td>Revised for Simulink 9.3 (Release 2019a)</td>
</tr>
<tr>
<td>September 2019</td>
<td>Online only</td>
<td>Revised for Simulink 10.0 (Release 2019b)</td>
</tr>
</tbody>
</table>
Examples of Setting Model Parameters ............... 6-111

Common Block Properties .......................... 6-112
   About Common Block Properties ................ 6-112
   Examples of Setting Block Properties .......... 6-131

Block-Specific Parameters .......................... 6-132
   Programmatic Parameters of Blocks and Models .... 6-132
   Block-Specific Parameters and Programmatic Equivalents . 6-133

Mask Parameters .................................... 6-281
   About Mask Parameters .......................... 6-281

Tools and Apps — Alphabetical List

Objects — Alphabetical List

Fixed-Point Tool

Fixed-Point Tool Parameters and Dialog Box ........... 9-2
   Main Toolbar ..................................... 9-2
   Model Hierarchy Pane ............................ 9-5
   Contents Pane .................................... 9-5
   Customizing the Contents Pane View ............... 9-8
   Dialog Pane ...................................... 9-10
   Fixed-Point Advisor .............................. 9-12
   Configure model settings ........................ 9-13
   Run name ........................................ 9-14
   Simulate ........................................ 9-15
   Merge instrumentation results from multiple simulations . 9-15
   Derive ranges for selected system ................ 9-16
Propose .................................................. 9-16
Propose for ........................................... 9-17
Default fraction length ............................... 9-17
Default word length .................................. 9-18
When proposing types use ............................ 9-18
Safety margin for simulation min/max (%) .......... 9-18

Advanced Settings ......................................... 9-20
  Advanced Settings Overview .......................... 9-20
  Fixed-point instrumentation mode .................. 9-20
  Data type override ................................... 9-21
  Data type override applies to ....................... 9-23
  Name of shortcut ..................................... 9-25
  Allow modification of fixed-point instrumentation settings ... 9-25
  Allow modification of data type override settings .... 9-26
  Allow modification of run name ....................... 9-27
  Run name ............................................. 9-27
  Capture system settings ............................... 9-27
  Fixed-point instrumentation mode .................. 9-27
  Data type override ................................... 9-28
  Data type override applies to ....................... 9-29

Model Advisor Checks

Simulink Checks ........................................... 10-2
  Simulink Check Overview ............................. 10-4
  Migrating to Simplified Initialization Mode Overview .... 10-5
  Identify unconnected lines, input ports, and output ports ... 10-5
  Check root model Inport block specifications .............. 10-6
  Check optimization settings .......................... 10-7
  Check diagnostic settings ignored during accelerated model 
    reference simulation ................................ 10-10
  Check for parameter tunability information ignored for 
    referenced models .................................. 10-10
  Check for implicit signal resolution .................... 10-11
  Check for optimal bus virtuality ...................... 10-12
  Check for Discrete-Time Integrator blocks with initial condition 
    uncertainty ........................................ 10-13
  Identify disabled library links ........................ 10-14
Check for large number of function arguments from virtual bus across model reference boundary .................................. 10-15
Identify parameterized library links .................................. 10-15
Identify unresolved library links .................................. 10-16
Identify configurable subsystem blocks for converting to variant subsystem blocks .................................. 10-17
Identify Variant Model blocks and convert those to Variant Subsystem containing Model block choices .................. 10-18
Identify Variant blocks using Variant objects with empty conditions .................................. 10-18
Check usage of function-call connections ............................ 10-19
Check Data Store Memory blocks for multitasking, strong typing, and shadowing issues .................................. 10-19
Check if read/write diagnostics are enabled for data store blocks .................................. 10-21
Check data store block sample times for modeling errors .................................. 10-22
Check for potential ordering issues involving data store access .................................. 10-23
Check structure parameter usage with bus signals .......................... 10-24
Check Delay, Unit Delay and Zero-Order Hold blocks for rate transition .................................. 10-26
Check for calls to slDataTypeAndScale .................................. 10-28
Check bus signals treated as vectors .................................. 10-29
Check for potentially delayed function-call subsystem return values .................................. 10-30
Identify block output signals with continuous sample time and non-floating point data type .................................. 10-31
Check usage of Merge blocks .................................. 10-32
Check usage of Outport blocks .................................. 10-35
Check usage of Discrete-Time Integrator blocks .............. 10-46
Check model settings for migration to simplified initialization mode .................................. 10-47
Check S-functions in the model .................................. 10-49
Check for non-continuous signals driving derivative ports .................................. 10-50
Runtime diagnostics for S-functions .................................. 10-51
Check model for foreign characters .................................. 10-52
Identify unit mismatches in the model .................................. 10-53
Identify automatic unit conversions in the model .................. 10-53
Identify disallowed unit systems in the model .................. 10-54
Identify undefined units in the model .................................. 10-54
Check model for block upgrade issues .................................. 10-55
Check model for block upgrade issues requiring compile time information .................................. 10-56
Check that the model is saved in SLX format .................. 10-57
Check model for SB2SL blocks .................................. 10-58
Check Model History properties .................................. 10-58
Identify Model Info blocks that can interact with external source control tools .............................................. 10-59
Identify Model Info blocks that use the Configuration Manager ................................................................. 10-60
Check model for legacy 3DoF or 6DoF blocks ............... 10-61
Check model and local libraries for legacy Aerospace Blockset blocks ................................................................. 10-62
Check model for Aerospace Blockset navigation blocks .......... 10-62
Check and update masked blocks in library to use promoted parameters .......................................................... 10-63
Check and update mask image display commands with unnecessary imread() function calls .............................. 10-64
Check and update mask to affirm icon drawing commands dependency on mask workspace ............................... 10-65
Identify masked blocks that specify tabs in mask dialog using MaskTabNames parameter ..................................... 10-66
Identify questionable operations for strict single-precision design ........................................................................ 10-67
Check get_param calls for block CompiledSampleTime ........ 10-68
Check if all simulation outputs are returned as a single Simulink.SimulationOutput object .................................... 10-70
Check model for parameter initialization and tuning issues ......................................................................................... 10-71
Check for virtual bus across model reference boundaries .. 10-72
Check model for custom library blocks that rely on frame status of the signal .......................................................... 10-74
Check model for S-function upgrade issues ....................... 10-75
Check Rapid accelerator signal logging ............................... 10-76
Check virtual bus inputs to blocks ....................................... 10-77
Check for root outports with constant sample time ............ 10-81
Analyze model hierarchy and continue upgrade sequence ... 10-81
Check Access to Data Stores ........................................... 10-83
Check relative execution orders for Data Store Read and Data Store Write blocks ..................................................... 10-84
Model Reference Conversion Advisor

Check Conversion Input Parameters

Performance Advisor Checks

Simulink Performance Advisor Checks

Baseline
Checks that Require Update Diagram
Checks that Require Simulation to Run
Check Simulation Modes Settings
Check Compiler Optimization Settings
Create baseline
Identify resource-intensive diagnostic settings
Check optimization settings
Identify inefficient lookup table blocks
Check MATLAB System block simulation mode
Identify Interpreted MATLAB Function blocks
Identify simulation target settings
Check model reference rebuild setting
Identify Scope blocks
Identify active instrumentation settings on the model
Check model reference parallel build
Check Delay block circular buffer setting
Check continuous and discrete rate coupling
Check zero-crossing impact on continuous integration
Check discrete signals driving derivative port
Check solver type selection
Select multi-thread co-simulation setting on or off
Identify co-simulation signals for numerical compensation
Select simulation mode
Select compiler optimizations on or off
Final Validation
# Simulink Limits

## Maximum Size Limits of Simulink Models

### Systems and Models

- **System**
- **System Component**
- **Model**
- **Model Component**
- **Differential Algebraic Equations**

### Simulink Models

- **Block Diagram**
- **Blocks**
- **Lines**
- **Data**
- **Parameters**
- **Property**
- **State variable**
- **Sample Time**
- **Units**
- **Direct Feedthrough**
- **Algebraic Loop**
- **Artificial Algebraic Loops**
- **Zero-Crossing Detection**

### Simulink Simulation

- **Compilation**
- **Callback**
- **Execution Order**
- **Simulation**
- **Solver**

### Simulink Tools

- **Programming Constructs in Simulink**

### Model Development Processes

- **Test Harness**

---

*xi*
Create Bus Ports in a Subsystem ........................................ 14-7
Manage Bus-to-Vector Conversions ................................. 14-10
Assign Signal Values to a Bus ........................................ 14-13
Initialize Your Model Using the Callback Button Block .... 14-14
Control a Parameter Value with Callback Button Blocks .. 14-15
Control the Duty Cycle of a PWM Signal Using Dashboard Blocks .................................................. 14-17
Control Merging Signals with the Push Button Block ...... 14-19
Tune the Relative Slip for an Anti-Lock Braking System ... 14-22
Interactively Simulate a Vehicle Climate Control System . 14-24
Interactively Simulate a Thermal Model of a House ...... 14-27
Create a Realistic Dashboard Using the Custom Gauge Block .......................................................... 14-30
Solve a Linear System of Algebraic Equations ............... 14-34
Model a Planar Pendulum ................................. 14-35
Improved Linearization with Transfer Fcn Blocks .......... 14-39
View Dead Zone Output on Sine Wave ....................... 14-40
View Backlash Output on Sine Wave ......................... 14-42
Prelookup With External Breakpoint Specification .......... 14-44
Prelookup with Evenly Spaced Breakpoints ................. 14-45
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Configure the Prelookup Block to Output Index and Fraction as a Bus</td>
<td>14-46</td>
</tr>
<tr>
<td>Approximating the sinh Function Using the Lookup Table Dynamic Block</td>
<td>14-48</td>
</tr>
<tr>
<td>Create a Logarithm Lookup Table</td>
<td>14-50</td>
</tr>
<tr>
<td>Providing Table Data as an Input to the Direct Lookup Table Block</td>
<td>14-51</td>
</tr>
<tr>
<td>Specifying Table Data in the Direct Lookup Table Block Dialog Box</td>
<td>14-52</td>
</tr>
<tr>
<td>Using the Quantizer and Saturation blocks in sldemo_boiler</td>
<td>14-53</td>
</tr>
<tr>
<td>Scalar Expansion with the Coulomb and Viscous Friction Block</td>
<td>14-54</td>
</tr>
<tr>
<td>Sum Block Reorders Inputs</td>
<td>14-55</td>
</tr>
<tr>
<td>Iterated Assignment with the Assignment Block</td>
<td>14-57</td>
</tr>
<tr>
<td>View Sample Time Using the Digital Clock Block</td>
<td>14-58</td>
</tr>
<tr>
<td>Bit Specification Using a Positive Integer</td>
<td>14-59</td>
</tr>
<tr>
<td>Bit Specification Using an Unsigned Integer Expression</td>
<td>14-60</td>
</tr>
<tr>
<td>Track Running Minimum Value of Chirp Signal</td>
<td>14-61</td>
</tr>
<tr>
<td>Horizontal Matrix Concatenation</td>
<td>14-63</td>
</tr>
<tr>
<td>Vertical Matrix Concatenation</td>
<td>14-64</td>
</tr>
<tr>
<td>Multidimensional Matrix Concatenation</td>
<td>14-65</td>
</tr>
<tr>
<td>Unary Minus of Matrix Input</td>
<td>14-66</td>
</tr>
<tr>
<td>Sample Time Math Operations Using the Weighted Sample Time Math Block</td>
<td>14-67</td>
</tr>
</tbody>
</table>
Construct Complex Signal from Real and Imaginary Parts ................................................. 14-68
Construct Complex Signal from Magnitude and Phase Angle ................................................. 14-69
Find Nonzero Elements in an Array ......................................................................................... 14-70
Calculate the Running Minimum Value with the MinMax Running Resettable Block ................. 14-71
Find Maximum Value of Input .................................................................................................... 14-73
Permute Array Dimensions ....................................................................................................... 14-75
Multiply Inputs of Different Dimensions with the Product Block ............................................ 14-76
Multiply and Divide Inputs Using the Product Block ................................................................. 14-77
Divide Inputs of Different Dimensions Using the Divide Block ................................................... 14-78
Complex Division Using the Product of Elements Block ......................................................... 14-79
Element-Wise Multiplication and Division Using the Product of Elements Block ......................... 14-80
sin Function with Floating-Point Input ....................................................................................... 14-81
sincos Function with Fixed-Point Input ..................................................................................... 14-82
Trigonometric Function Block Behavior for Complex Exponential Output ............................... 14-83
Create Bus Object from Constant Block .................................................................................. 14-84
Control Algorithm Execution Using Enumerated Signal ......................................................... 14-85
Integer and Enumerated Data Type Support in the Ground Block ........................................... 14-87
<table>
<thead>
<tr>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fixed-Point Data Type Support in the Ground Block</td>
<td>14-88</td>
</tr>
<tr>
<td>Read 1-D Array and Structure From Workspace</td>
<td>14-89</td>
</tr>
<tr>
<td>Read Structure From Workspace Using Model Sample Time</td>
<td>14-90</td>
</tr>
<tr>
<td>Read 2-D Signals in Structure Format From Workspace</td>
<td>14-92</td>
</tr>
<tr>
<td>From File Block Loading Timeseries Data</td>
<td>14-93</td>
</tr>
<tr>
<td>Eliminate Singleton Dimension with the Squeeze Block</td>
<td>14-94</td>
</tr>
<tr>
<td>Difference Between Time- and Sample-Based Pulse Generation</td>
<td>14-95</td>
</tr>
<tr>
<td>Specify a Waveform with the Repeating Sequence Block</td>
<td>14-97</td>
</tr>
<tr>
<td>Tune Phase Delay on Pulse Generator During Simulation</td>
<td>14-99</td>
</tr>
<tr>
<td>Difference Sine Wave Signal</td>
<td>14-100</td>
</tr>
<tr>
<td>Discrete-Time Derivative of Floating-Point Input</td>
<td>14-102</td>
</tr>
<tr>
<td>First-Order Sample-and-Hold of a Sine Wave</td>
<td>14-104</td>
</tr>
<tr>
<td>Calculate and Display Simulation Step Size using Memory and Clock Blocks</td>
<td>14-106</td>
</tr>
<tr>
<td>Capture the Velocity of a Bouncing Ball with the Memory Block</td>
<td>14-107</td>
</tr>
<tr>
<td>Implement a Finite-State Machine with the Combinatorial Logic and Memory Blocks</td>
<td>14-109</td>
</tr>
<tr>
<td>Discrete-Time Integration Using the Forward Euler Integration Method</td>
<td>14-112</td>
</tr>
<tr>
<td>Signal Routing with the From, Goto, and Goto Tag Visibility Blocks</td>
<td>14-113</td>
</tr>
</tbody>
</table>
Zero-Based and One-Based Indexing with the Index Vector Block .................................................. 14-116

Noncontiguous Values for Data Port Indices of Multiport Switch Block ........................................ 14-117

Using Variable-Size Signals on the Delay Block ........ 14-118

Bus Signals with the Delay Block for Frame-Based Processing ....................................................... 14-120

Control Execution of Delay Block with Enable Port ..... 14-121

Zero-Based Indexing for Multiport Switch Data Ports .... 14-123

One-Based Indexing for Multiport Switch Data Ports .... 14-124

Enumerated Names for Data Port Indices of the Multiport Switch Block ........................................ 14-126

Prevent Block Windup in Multiloop Control .............. 14-127

Bumpless Control Transfer .................................. 14-128

Bumpless Control Transfer with a Two-Degree-of-Freedom PID Controller .................................... 14-129

Using a Bit Set block ............................................ 14-130

Using a Bit Clear block ........................................... 14-131

Two-Input AND Logic ........................................... 14-132

Circuit Logic ..................................................... 14-133

Unsigned Inputs for the Bitwise Operator Block ......... 14-134

Signed Inputs for the Bitwise Operator Block .......... 14-135

Merge Block with Input from Atomic Subsystems ....... 14-136

Index Options with the Selector Block .................. 14-137
Switch Block with a Boolean Control Port Example ...... 14-139
Merge Block with Unequal Input Widths Example ...... 14-140
Detect Rising Edge of Signals ................................ 14-143
Detect Falling Edge Using the Detect Fall Nonpositive Block
                                          .............................................. 14-145
Detect Increasing Signal Values with the Detect Increase Block
                                          .............................................. 14-147
Extract Bits from Stored Integer Value .................. 14-149
Detect Signal Values Within a Dynamically Specified Interval
                                          .............................................. 14-150
Model a Digital Thermometer Using the Polynomial Block
                                          .............................................. 14-152
Convert Data Types in Simulink Models ................. 14-153
Control Data Types with the Data Type Duplicate Block .... 14-155
Probe Sample Time of a Signal .............................. 14-156
Convert Signals Between Continuous Time and Discrete Time
                                          .............................................. 14-157
Convert Muxed Signal to a Vector ......................... 14-159
Create Contiguous Copy of a Bus Signal ................. 14-160
Convert Virtual Bus to a Nonvirtual Bus ................. 14-161
Convert Nonvirtual Bus to Virtual Bus ................... 14-162
Remove Scaling from a Fixed-Point Signal ................ 14-163
Stop Simulation Block with Relational Operator Block .... 14-164
Output Simulation Data with Blocks ...................... 14-165
<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Increment and Decrement Real-World Values</td>
<td>14-170</td>
</tr>
<tr>
<td>Increment and Decrement Stored Integer Values</td>
<td>14-173</td>
</tr>
<tr>
<td>Specify a Vector of Initial Conditions for a Discrete Filter Block</td>
<td>14-174</td>
</tr>
<tr>
<td>Generate Linear Models for a Rising Edge Trigger Signal</td>
<td>14-176</td>
</tr>
<tr>
<td>Generate Linear Models at Predetermined Times</td>
<td>14-178</td>
</tr>
<tr>
<td>Capture Measurement Descriptions in a DocBlock</td>
<td>14-180</td>
</tr>
<tr>
<td>Square Root of Negative Values</td>
<td>14-181</td>
</tr>
<tr>
<td>Signed Square Root of Negative Values</td>
<td>14-182</td>
</tr>
<tr>
<td>rSqrt of Floating-Point Inputs</td>
<td>14-183</td>
</tr>
<tr>
<td>rSqrt of Fixed-Point Inputs</td>
<td>14-184</td>
</tr>
<tr>
<td>Model a Series RLC Circuit</td>
<td>14-185</td>
</tr>
<tr>
<td>Extract Vector Elements and Distribute Evenly Across Outputs</td>
<td>14-188</td>
</tr>
<tr>
<td>Extract Vector Elements Using the Demux Block</td>
<td>14-189</td>
</tr>
<tr>
<td>Detect Change in Signal Values</td>
<td>14-191</td>
</tr>
<tr>
<td>Detect Fall to Negative Signal Values</td>
<td>14-193</td>
</tr>
<tr>
<td>Detect Decreasing Signal Values</td>
<td>14-195</td>
</tr>
<tr>
<td>Function-Call Blocks Connected to Branches of the Same Function-Call Signal</td>
<td>14-197</td>
</tr>
<tr>
<td>Function-Call Feedback Latch on Feedback Signal Between Child and Parent</td>
<td>14-198</td>
</tr>
<tr>
<td>Single Function-Call Subsystem</td>
<td>14-199</td>
</tr>
</tbody>
</table>
Function-Call Subsystem with Merged Signal As Input . . . 14-200
Partitioning an Input Signal with the For Each Block . . . 14-201
Specifying the Concatenation Dimension in the For Each Block ............................................. 14-202
Working with the Initialize Function, Reset Function, and Terminate Function Blocks ............................................. 14-203
Reading and Writing States with the Initialize Function and Terminate Function Blocks ............................................. 14-204

Simulink Featured Examples

Simulation of a Bouncing Ball ....................................................... 15-7
Single Hydraulic Cylinder Simulation ............................................. 15-17
Thermal Model of a House ............................................................. 15-31
Approximating Nonlinear Relationships: Type S Thermocouple ..................................................... 15-38
Digital Waveform Generation: Approximating a Sine Wave ..................................................... 15-48
Accurate Zero-Crossing Detection .................................................. 15-61
Spiral Galaxy Formation Simulation Using MATLAB Function Blocks ..................................................... 15-63
Modulo-4 Counter Using Flip-Flops ............................................... 15-68
Counters Using Conditionally Executed Subsystems ..................................................... 15-70
Friction Model with Hard Stops ..................................................... 15-73
State Events ......................................................... 15-75
Bang-Bang Control Using Temporal Logic ......................... 15-77
Inverted Pendulum with Animation .................................. 15-78
Double Spring Mass System ........................................... 15-80
Tank Fill and Empty with Animation ................................ 15-82
Simulating Systems with Variable Transport Delay Phenomena
.................................................................................. 15-85
Modeling a Foucault Pendulum .......................................... 15-90
Foucault Pendulum Model with VRML Visualization ............. 15-101
Exploring Variable-Step Solvers Using a Stiff Model .............. 15-104
Exploring the Solver Jacobian Structure of a Model .............. 15-111
Double Bouncing Ball: Use of Adaptive Zero-Crossing Location
.................................................................................. 15-122
Four Hydraulic Cylinder Simulation .................................... 15-132
Two Cylinder Model with Load Constraints .......................... 15-139
Modeling a Fault-Tolerant Fuel Control System .................. 15-149
Using a Data Dictionary to Manage the Data for a Fuel Control
System ........................................................................ 15-168
Modeling Engine Timing Using Triggered Subsystems .......... 15-172
Engine Timing Model with Closed Loop Control .................. 15-184
Building a Clutch Lock-Up Model ...................................... 15-189
Modeling Clutch Lock-Up Using If Blocks ........................... 15-203
Modeling an Anti-Lock Braking System ............................... 15-210
Automotive Suspension ........................................ 15-219
Modeling an Automatic Transmission Controller ........ 15-226
Vehicle Electrical System ..................................... 15-240
Simulating Automatic Climate Control Systems .......... 15-242
Vehicle Electrical and Climate Control Systems .......... 15-249
Power Window Control Project .............................. 15-257
Developing the Apollo Lunar Module Digital Autopilot ... 15-266
Designing a High Angle of Attack Pitch Mode Control ... 15-273
Six Degrees of Freedom (6-DoF) Motion Platform .......... 15-291
Aircraft Longitudinal Flight Control ....................... 15-294
Radar Tracking Using MATLAB Function Block .......... 15-297
Optical Sensor Image Generation ............................ 15-299
Air Traffic Control Radar Design ............................ 15-311
Designing a Guidance System in MATLAB and Simulink .. 15-318
Airframe Trim and Linearize ................................ 15-336
Anti-Windup Control Using a PID Controller ............. 15-341
Bumpless Control Transfer Between Manual and PID Control
.............................................................................. 15-359
Two Degree-of-Freedom PID Control for Setpoint Tracking
.............................................................................. 15-371
Data Typing in Simulink ..................................... 15-379
Data Typing Filter .............................................. 15-383
xxi
Simulink Bus Signals .......................................................... 15-385
Modeling Arrays of Buses .................................................. 15-398
Matrix Signals ................................................................. 15-400
Variable-Size Signal Basic Operations ............................... 15-401
Variable-Size Signal Length Adaptation ............................ 15-403
Multimode Variable-Size Signal .......................................... 15-404
Parallel Channel Power Allocation .................................... 15-405
Merging Signals ............................................................... 15-407
Migration to Structure Parameters .................................... 15-425
Share Data Store Between Instances of a Reusable Algorithm .......................................................... 15-432
Attaching Input Data to External Inputs via Custom Input Mappings .................................................. 15-438
Using Mapping Modes with Custom-Mapped External Inputs .................................................. 15-442
Converting Harness-Driven Models to Use Harness-Free External Inputs .................................................. 15-448
Logging States in Structure Format ................................... 15-455
Saving and Restoring Simulation Operating Point ............... 15-459
Logging Intervals ............................................................... 15-464
Working with Big Data ...................................................... 15-468
Simulink Subsystem Semantics .......................................... 15-474
If-Then-Else Blocks .......................................................... 15-476
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Triggered Subsystems</td>
<td>15-479</td>
</tr>
<tr>
<td>Enabled Subsystems</td>
<td>15-482</td>
</tr>
<tr>
<td>Advanced Enabled Subsystems</td>
<td>15-485</td>
</tr>
<tr>
<td>Resettable Subsystems</td>
<td>15-489</td>
</tr>
<tr>
<td>Discrete and Continuous Resettable Subsystems</td>
<td>15-492</td>
</tr>
<tr>
<td>Variant Subsystems</td>
<td>15-495</td>
</tr>
<tr>
<td>Block Priority</td>
<td>15-501</td>
</tr>
<tr>
<td>Monitoring Ink Status on a Shared Printer Using Simulink Functions</td>
<td>15-502</td>
</tr>
<tr>
<td>Modeling Services and Access from Application Software</td>
<td>15-504</td>
</tr>
<tr>
<td>Modeling Reusable Components Using Multiply Instanced Simulink Functions</td>
<td>15-506</td>
</tr>
<tr>
<td>Dynamic Priority Scheduling of Functions</td>
<td>15-509</td>
</tr>
<tr>
<td>Component-Based Modeling with Model Reference</td>
<td>15-510</td>
</tr>
<tr>
<td>Viewing Signals in Model Reference Instances</td>
<td>15-515</td>
</tr>
<tr>
<td>Visualizing Model Reference Architectures</td>
<td>15-525</td>
</tr>
<tr>
<td>Introduction to Managing Data with Model Reference</td>
<td>15-538</td>
</tr>
<tr>
<td>Detailed Workflow for Managing Data with Model Reference</td>
<td>15-553</td>
</tr>
<tr>
<td>Interface Specification Using Bus Objects</td>
<td>15-576</td>
</tr>
<tr>
<td>Converting Subsystems to Model Reference</td>
<td>15-583</td>
</tr>
<tr>
<td>Using Data Stores Across Multiple Models</td>
<td>15-587</td>
</tr>
<tr>
<td>Model Reference Function-Call</td>
<td>15-595</td>
</tr>
</tbody>
</table>
Protected Models for Model Reference .................. 15-598
Model Reference Variants ............................... 15-605
Assigning Tasks to Cores for Multicore Programming ... 15-610
Implement an FFT on a Multicore Processor and an FPGA 15-613
Multicore Programming of a Field-Oriented Control on Zynq 15-619
Multicore Deployment of a Plant Model .................. 15-627
Modeling Objects with Identical Dynamics Using For Each Subsystem 15-633
Vectorizing a Scalar Algorithm with For Each Subsystem 15-639
Tiled Processing of 2D Signals with For Each Subsystem 15-641
Neighborhood Processing Using For Each Subsystems ... 15-643
Using a Project with SVN ............................... 15-647
Using a Project with Git ............................... 15-652
Using a Project ........................................ 15-657
Perform Impact Analysis with a Project .................. 15-662
Referencing Projects from Another Project ................ 15-665
Automate Label Management in a Project .................. 15-666
Running Custom Tasks with a Project .................... 15-670
Upgrade Simulink Models Using a Project ................ 15-674
Convert from MDL to SLX in a Project and Preserve Revision History 15-677
Export a Subset of a Project Using an Export Profile ........ 15-683
Creating Projects Programmatically ............................ 15-685
Compare and Merge Simulink Models ............................ 15-687
Compare and Merge Simulink Models Containing Stateflow
........................................................................ 15-691
Resolve Conflicts with Simulink Three-Way Merge .......... 15-695
Call C Functions Using C Caller Block .......................... 15-699
Bring Custom Image Filter Algorithms as Reusable Blocks in
Simulink .................................................................. 15-704
Custom Code and Hand Coded Blocks using the S-function API
........................................................................ 15-707
Inputs Passed by Value or Address to Legacy Functions .... 15-708
Output Passed by Return Argument from Legacy Functions
........................................................................ 15-713
Fixed Point Signals in Legacy Functions ....................... 15-716
Fixed Point Parameters in Legacy Functions .................. 15-719
Lookup Tables Implemented in Legacy Functions .......... 15-723
Start and Terminate Actions Within Legacy Functions ...... 15-727
Using Buses with Legacy Functions Having Structure
Arguments .................................................................. 15-732
Inherited Signal Dimensions for Legacy Function Arguments
........................................................................ 15-737
C++ Object Methods as Legacy Functions .................... 15-741
Persistent Memory Within Legacy Functions .................. 15-745
Multi-Dimensional Signals in Legacy Functions .......... 15-749
Complex Signals in Legacy Function .......................... 15-752
Specified or Inherited Sample Time with Legacy Functions .............................................. 15-755
System Identification for an FIR System Using MATLAB System Blocks ............................................ 15-759
MATLAB System Block with Variable-Size Input and Output Signals .................................................. 15-762
Illustration of Law of Large Numbers ..................... 15-764
Using Buses with MATLAB System Blocks ............... 15-766
Run Quality Checks on S-Functions .......................... 15-768
Interpolation Algorithm Selection and Trajectory Animation .............................................................. 15-770
Using the Prelookup and Interpolation Blocks ............ 15-773
Saving Memory in Prelookup and Interpolation Blocks by Using Smaller Data .................................... 15-780
Model Advisor ................................................... 15-781
Simulink Model Discretizer ..................................... 15-783
Introduction to Profiling Models .............................. 15-784
Introduction to Accelerating Models ........................ 15-788
Determining Why Simulink Accelerator Is Regenerating Code .............................................................. 15-792
Parallel Simulations Using Parsim: Test-Case Sweep .... 15-799
Parallel Simulations Using Parsim: Parameter Sweep in Normal Mode ............................................. 15-804
Parallel Simulations Using Parsim: Parameter Sweep in Rapid Accelerator Mode ............................................. 15-809

Rapid Accelerator Simulations Using Parsim .................. 15-814

Multiple Simulations Workflow Tips ............................ 15-819

Streamline Simulink Blockset Authoring Process with Blockset Designer ................................................................. 15-824

Importing a Co-Simulation FMU into Simulink ............ 15-826

Importing a Model Exchange FMU into Simulink .......... 15-828

Using Bus Signals and Structure Parameters in the FMU Import Block ................................................................. 15-830

Co-Simulation Signal Compensation ........................... 15-833

Multithread Co-Simulation .......................................... 15-836

Pulse Width Modulation Using MATLAB System Block ... 15-838

Modeling Cyber-Physical Systems ............................... 15-840

Schedule an Export-Function Model Using the Schedule Editor .................................................................................. 15-847
Blocks — Alphabetical List
Abs

Output absolute value of input

Library: Simulink / Math Operations
HDL Coder / HDL Floating Point Operations
HDL Coder / Math Operations

Description

The Abs block outputs the absolute value of the input.

For signed-integer data types, the absolute value of the most negative value is not representable by the data type. In this case, the Saturate on integer overflow check box controls the behavior of the block.

<table>
<thead>
<tr>
<th>If you...</th>
<th>The block...</th>
<th>And...</th>
</tr>
</thead>
</table>
| Select this check box | Saturates to the most positive value of the integer data type | • For 8-bit signed integers, -128 maps to 127.  
• For 16-bit signed integers, -32768 maps to 32767.  
• For 32-bit signed integers, -2147483648 maps to 2147483647. |
| Do not select this check box | Wraps to the most negative value of the integer data type | • For 8-bit signed integers, -128 remains -128.  
• For 16-bit signed integers, -32768 remains -32768.  
• For 32-bit signed integers, -2147483648 remains -2147483648. |

The Abs block supports zero-crossing detection. However, when you select Enable zero-crossing detection on the dialog box, the block does not report the simulation minimum or maximum in the Fixed-Point Tool. If you want to use the Fixed-Point Tool to analyze a model, disable zero-crossing detection for all Abs blocks in the model first.
Ports

Input

Port_1 — Input signal
scalar | vector

Input signal to the absolute value block.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

Output

Port_1 — Absolute value output signal
scalar | vector

Absolute value of the input signal.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

Parameters

Main

Enable zero-crossing detection — Enable zero-crossing detection
on (default) | Boolean

Select to enable zero-crossing detection. For more information, see “Zero-Crossing Detection” in the Simulink documentation.

Programmatic Use
Block Parameter: ZeroCross
Type: character vector, string
Values: 'off' | 'on'
Default: 'on'

Sample time — Specify sample time as a value other than -1
-1 (default) | scalar | vector
Specify the sample time as a value other than -1. For more information, see “Specify Sample Time”.

**Dependencies**

This parameter is not visible unless it is explicitly set to a value other than -1. To learn more, see “Blocks for Which Sample Time Is Not Recommended”.

**Programmatic Use**

**Block Parameter:** SampleTime  
Type: character vector  
Values: scalar or vector  
Default: ' -1'

**Signal Attributes**

Click the **Show data type assistant** button to display the **Data Type Assistant**, which helps you set the data type attributes. For more information, see “Specify Data Types Using Data Type Assistant”.

**Output minimum — Minimum output value for range checking**

[-] (default) | scalar

Lower value of the output range that Simulink checks.

Simulink uses the minimum to perform:

- Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters”) for some blocks.
- Simulation range checking (see “Signal Ranges” and “Enable Simulation Range Checking”).
- Automatic scaling of fixed-point data types.
- Optimization of the code that you generate from the model. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. For more information, see “Optimize using the specified minimum and maximum values” (Simulink Coder).

**Note** **Output minimum** does not saturate or clip the actual output signal. Use the Saturation block instead.
Programmatic Use
Block Parameter: OutMin
Type: character vector
Values: '[ ]' | scalar
Default: '[ ]'

Output maximum — Maximum output value for range checking
[ ] (default) | scalar

Upper value of the output range that Simulink checks.

Simulink uses the maximum value to perform:

- Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters”) for some blocks.
- Simulation range checking (see “Signal Ranges” and “Enable Simulation Range Checking”).
- Automatic scaling of fixed-point data types.
- Optimization of the code that you generate from the model. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. For more information, see “Optimize using the specified minimum and maximum values” (Simulink Coder).

Note Output maximum does not saturate or clip the actual output signal. Use the Saturation block instead.

Programmatic Use
Block Parameter: OutMax
Type: character vector
Values: '[ ]' | scalar
Default: '[ ]'

Output data type — Specify the output data type
Inherit: Same as input (default) | Inherit: Inherit via back propagation | double | single | int8 | int16 | int32 | uint8 | uint16 | uint32 | int64 | uint64 | fixdt(1,16,2^0,0) | <data type expression> | ...

Choose the data type for the output. The type can be inherited, specified directly, or expressed as a data type object such as Simulink.NumericType.
Programmatic Use
Block Parameter: OutDataTypeStr
Type: character vector
Values: 'Inherit: Same as input', 'Inherit: Inherit via back propagation', 'single', 'int8', 'uint8', 'int16', 'uint16', 'int32', 'uint32', 'int64', 'uint64', fixdt(1,16,0), fixdt(1,16,2^0,0), fixdt(1,16,2^0,0).
'<data type expression>'
Default: 'Inherit: Same as input'

Lock output data type setting against changes by the fixed-point tools — Prevent fixed-point tools from overriding Output data type
off (default) | on

Select this parameter to prevent the fixed-point tools from overriding the Output data type you specify on the block. For more information, see “Use Lock Output Data Type Setting” (Fixed-Point Designer).

Programmatic Use
Block Parameter: LockScale
Type: character vector
Values: 'off' | 'on'
Default: 'off'

Integer rounding mode — Rounding mode for fixed-point operations
Floor (default) | Ceiling | Convergent | Nearest | Round | Simplest | Zero

Specify the rounding mode for fixed-point operations. For more information, see “Rounding” (Fixed-Point Designer).

Block parameters always round to the nearest representable value. To control the rounding of a block parameter, enter an expression using a MATLAB® rounding function into the mask field.

Programmatic Use
Block Parameter: RndMeth
Type: character vector
Values: 'Ceiling' | 'Convergent' | 'Floor' | 'Nearest' | 'Round' | 'Simplest' | 'Zero'
Default: 'Floor'

Saturate on integer overflow — Choose the behavior when integer overflow occurs
off (default) | on
When you select this check box, saturation applies to every internal operation on the block, not just the output or result. In general, the code generation process can detect when overflow is not possible. In this case, the code generator does not produce saturation code.

<table>
<thead>
<tr>
<th>Action</th>
<th>Reason for Taking This Action</th>
<th>What Happens</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Select this check box.</td>
<td>Your model has possible overflow and you want explicit saturation protection in the generated code.</td>
<td>Overflows saturate to the maximum value that the data type can represent.</td>
<td>The number 130 does not fit in a signed 8-bit integer and saturates to 127.</td>
</tr>
<tr>
<td>Do not select this check box.</td>
<td>You want to optimize efficiency of your generated code.</td>
<td>Overflows wrap to the appropriate value that is representable by the data type.</td>
<td>The number 130 does not fit in a signed 8-bit integer and wraps to -126.</td>
</tr>
</tbody>
</table>

**Programmatic Use**

**Block Parameter:** SaturateOnIntegerOverflow  
**Type:** character vector  
**Value:** 'off' | 'on'  
**Default:** 'off'

**Block Characteristics**

<table>
<thead>
<tr>
<th>Data Types</th>
<th>double</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder™ provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture
This block has a single, default HDL architecture.

HDL Block Properties

<table>
<thead>
<tr>
<th>General</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ConstrainedOutputPipeline</td>
<td>Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. For more details, see “ConstrainedOutputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td>InputPipeline</td>
<td>Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “InputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td>OutputPipeline</td>
<td>Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “OutputPipeline” (HDL Coder).</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Native Floating Point</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>LatencyStrategy</td>
<td>Specify whether to map the blocks in your design to inherit, Max, Min, or Zero for the floating-point operator. The default is inherit. See also “LatencyStrategy” (HDL Coder).</td>
</tr>
</tbody>
</table>
**Complex Data Support**

This block does not support code generation for complex signals. To calculate the magnitude of a complex number, use the Complex to Magnitude-Angle HDL Optimized block instead.

**PLC Code Generation**
Generate Structured Text code using Simulink® PLC Coder™.

**Fixed-Point Conversion**
Design and simulate fixed-point systems using Fixed-Point Designer™.

**See Also**
Sign | Sum

**Introduced before R2006a**
Action Port

Add control port for action signal to subsystem
Library: Ports & Subsystems

Description

The Action Port block controls the execution of these subsystem blocks:

- If Action Subsystem blocks connected to If blocks.
- Switch Case Action Subsystem blocks connected to Switch Case blocks.
- Simulink based states in Stateflow® charts. See “Create and Edit Simulink Based States” (Stateflow).

Parameters

*States when execution is resumed — Select handling of internal states*

held (default) | reset

Select how to handle internal states when a subsystem with an Action Port block reenables.

held

When the subsystem reenables, retain the previous state values of the subsystem. Previous state values between calls are retained even if you call other subsystem blocks connected to the If or Switch Case block.

reset

When the subsystem reenables, reinitialize the state values.

A subsystem reenables when the logical expression for its action port evaluates to true after having been previously false. In the following example, the Action Port blocks for both subsystems A and B have the *States when execution is resumed* parameter set to *reset*.
When `case[1]` is true, subsystem A is executed. Repeated calls to subsystem A while `case [1]` continues to be true, does not reset its state values. The same behavior applies to subsystem B.

**Programmatic Use**

**Block Parameter:** InitializeStates  
**Type:** character vector  
**Value:** 'held' | 'reset'  
**Default:** 'held'

**Propagate sizes of variable-size signals — Select when to propagate a variable-size signal**  
Only when execution is resumed (default) | During execution

Select when to propagate a variable-size signal.

**Only when execution is resumed**  
Propagate variable-size signals only when reenabling the subsystem containing the Action Port block.

**During execution**  
Propagate variable-size signals at each time step.

**Programmatic Use**

**Block Parameter:** PropagateVarSize
**Type:** character vector  
**Values:** 'Only when execution is resumed' | 'During execution'  
**Default:** 'Only when execution is resumed'  

**See Also**  
If | If Action Subsystem | Switch Case | Switch Case Action Subsystem  

**Topics**  
Select Subsystem Execution  

**Introduced before R2006a**
Algebraic Constraint

Constrain input signal

Library: Simulink / Math Operations

Description

The Algebraic Constraint block constrains the input signal \( f(z) \) to \( z \) or 0 and outputs an algebraic state \( z \). The block outputs a value that produces 0 or \( z \) at the input. The output must affect the input through a direct feedback path. In other words, the feedback path only contains blocks with direct feedthrough. For example, you can specify algebraic equations for index 1 differential-algebraic systems (DAEs).

Ports

Input

\( f(z) \) — Input signal
real scalar or vector

Signal is subjected to the constraint \( f(z) = 0 \) or \( f(z) = z \) to solve the algebraic loop.

Data Types: double

Output

\( z \) — Output state
real scalar or vector

Solution to the algebraic loop when the input signal \( f(z) \) is subjected to the constraint \( f(z) = 0 \) or \( f(z) = z \).

Data Types: double
**Parameters**

**Constraint — Constraint on input signal**

\[ f(z) = 0 \text{ (default)} | f(z) = z \]

Type of constraint for which to solve. You can solve for \( f(z) = 0 \) or \( f(z) = z \)

**Programmatic Use**

**Block Parameter:** Constraint  
**Type:** character vector  
**Values:** 'f(z) = 0' | 'f(z) = z'  
**Default:** 'f(z) = 0'

**Solver — Algebraic Loop Solver**

auto (default) | Trust region | Line search

Choose between the Trust region [1], [2] or Line search [3] algorithms to solve the algebraic loop. By default this value is set to auto, which selects the solver based on the model configuration

**Programmatic Use**

**Block Parameter:** Solver  
**Type:** character vector  
**Values:** 'auto' | 'Trust region' | 'Line search'  
**Default:** 'auto'

**Tolerance — Solver Tolerance**

auto (default) | positive scalar

This option is visible when you explicitly specify a solver to be used (Trust region or Line Search) in the Solver dropdown menu. Specify a smaller value for higher accuracy or a larger value for faster execution. By default it is set to auto.

**Programmatic Use**

**Block Parameter:** Tolerance  
**Type:** character vector  
**Values:** 'auto' | positive scalar  
**Default:** 'auto'

**Initial Guess — Initial guess of solution value**

0 (default) | scalar
Initial guess for the algebraic state $z$ that is close to the expected solution value to improve the efficiency of the algebraic loop solver. By default, this value is set to 0

**Programmatic Use**

**Block Parameter:** InitialGuess  
**Type:** character vector  
**Values:** scalar  
**Default:** '0'

## Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>double</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>no</td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
</tr>
</tbody>
</table>

## References


Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

Not recommended for production code.

See Also
“Algebraic Loop Concepts”

Introduced before R2006a
Argument Inport

Argument input port for Simulink Function block
Library: User-Defined Functions

Description

This block is an argument input port for a function that you define in the Simulink Function block.

Ports

Input

\[ u \] — Argument input
scalar | vector | matrix

The Argument Inport block accepts complex or real signals of any data type that Simulink supports, including fixed-point and enumerated data types. The Argument Inport block also accepts a bus object as a data type.

The complexity and data type of the block output are the same as the argument input. You can specify the signal type and data type of an input argument to an Argument Inport block using the Signal type and Data type parameters.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus

Output

\[ \text{out} \] — Block output
scalar | vector | matrix

Block output signal from this block.
Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus

Parameters

Port number — Specify port number of block
1 (default) | integer

Specify the order in which the port that corresponds to the block appears in the parent subsystem or model block.

1
   Specify the first port location for this block.

integer
   Specify location of port.

Programmatic Use
Block parameter: Port
Type: character vector
Value: '1' | '<integer>'
Default: '1'

Argument name — Specify input argument name
u (default) | character vector

Specify input argument name for the function prototype displayed on the face of the Simulink Function block.

u
   Default name for the input argument.

character vector
   Name of the input argument.

Programmatic Use
Block parameter: ArgumentName
Type: character vector
Value: 'u' | '<character vector>'
Default: 'u'
Minimum — Specify minimum value for block output

Specify the minimum value for the block output signal.

**Note** If you specify a bus object as the data type for this block, do not set the minimum value for bus data on the block. Simulink ignores this setting. Instead, set the minimum values for bus elements of the bus object specified as the data type. For information on the Minimum property of a bus element, see `Simulink.BusElement`.

Simulink uses this value to perform Simulation range checking and automatic scaling of fixed-point data types.

[]

Minimum value not specified.

number

Finite real double scalar value.

**Programmatic Use**

**Block parameter:** `OutMin`

**Type:** character vector

**Value:** `'' | '<number>'`

**Default:** `''`

Maximum — Specify maximum value for block output

Specify the maximum value for the block output signal.

**Note** If you specify a bus object as the data type for this block, do not set the maximum value for bus data on the block. Simulink ignores this setting. Instead, set the maximum values for bus elements of the bus object specified as the data type. For information on the Maximum property of a bus element, see `Simulink.BusElement`.

Simulink uses this value to perform Simulation range checking and automatic scaling of fixed-point data types.
Maximum value not specified.

Finite real double scalar value.

**Programmatic Use**

**Block parameter:** OutMax

**Type:** character vector

**Value:** `['']` | `<number>`

**Default:** `['']`

**Data type — Specify block output data type**

double (default) | single | int8 | uint8 | int16 | uint16 | int32 | uint32 | int64 | uint64 | boolean | fixdt(1,16) | fixdt(1,16,0) | fixdt(1,16,2^,0) | Enum: <class name> | Bus: <object name> | <data type expression>

Specify the block output data type.

double
   Data type is double.

single
   Data type is single.

int8
   Data type is int8.

uint8
   Data type is uint8.

int16
   Data type is int16.

uint16
   Data type is uint16.

int32
   Data type is int32.

uint32
   Data type is uint32.
int64
  Data type is int64.
uint64
  Data type is uint64.
boolean
  Data type is boolean.
fixdt(1,16,0)
  Data type is fixed point fixdt(1,16,0).
fixdt(1,16,2^0,0)
  Data type is fixed point fixdt(1,16,2^0,0).
Enum: <class name>
  Data type is enumerated, for example, Enum: Basic Colors.
Bus: <object name>
  Data type is a bus object.
<data type expression>
  The name of a data type object, for example Simulink.NumericType

Do not specify a bus object as the expression.

**Programmatic Use**

**Block parameter:** OutDataTypeStr

**Type:** character vector

**Value:** 'double' | 'single' | 'int8' | 'uint8' | 'int16' | 'uint16' | 'int32' | 'uint32' | 'boolean' | 'fixdt(1,16)' | 'fixdt(1,16,0)' | 'fixdt(1,16,2^0,0)' | '<data type expression>'

**Default:** 'double'

**Lock output data type setting against changes by the fixed-point tools — Prevent fixed-point tools from overriding data type**

off (default) | on

Control changes to data type settings from the Fixed-Point Tool and the Fixed-Point Advisor. For more information, see “Use Lock Output Data Type Setting” (Fixed-Point Designer).
On
Locks all data type settings for this block.

Off
Allows the Fixed-Point Tool and the Fixed-Point Advisor to change data type settings for this block.

Programmatic Use
Block parameter: LockScale
Type: character vector
Value: 'off' | 'on'
Default: 'off'

Port dimensions — Specify port dimensions
1 (default) | n | [m n]
Specify the dimensions of the argument input signal to the block. For more information, see Outport.

1
Inherit port dimensions.

n
Vector signal of width n.

[m n]
Matrix signal having m rows and n columns.

Programmatic Use
Block parameter: PortDimensions
Type: character vector
Value: '1' | 'n' | '[m n']
Default: '1'

Signal type — Select real or complex signal
real (default) | complex
Select real or complex signal.

real
Specify the signal type as a real number.
complex
   Specify the signal type as a complex number.

**Programmatic Use**
**Block parameter:** SignalType
**Type:** character vector
**Value:** 'real' | 'complex'
**Default:** 'real'

**See Also**
Argument Outport | Function Caller | Simulink Function

**Topics**
“Simulink Functions Overview”
“Argument Specification for Simulink Function Blocks”

**Introduced in R2014b**
**Argument Outport**

Argument output port for Simulink Function block

**Library:** User-Defined Functions

**Description**

This block is an output argument port for a function that you define in the Simulink Function block.

**Ports**

**Input**

in — Block input

scalar | vector | matrix

Block input signal to this block.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus

**Output**

y — Argument output

scalar | vector | matrix

The Argument Outport block accepts real or complex signals of any data type that Simulink supports. An Argument Outport block can also accept fixed-point and enumerated data types when the block is not a root-level output port. The Argument Outport block also accepts a bus object as a data type.

The complexity and data type of the block input are the same as the argument output. You can specify the signal type and data type of an output argument from an Argument
Outport block using the **Signal type** and **Data type** parameters. For more information, see “Data Types Supported by Simulink”.

Data Types: `single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus`

**Parameters**

**Port number — Specify port number of block**

1 (default) | integer

Specify the order in which the port that corresponds to the block appears in the parent subsystem or model block.

1

Specify the first port location for this block.

integer

Specify location of port.

**Programmatic Use**

**Block parameter:** Port

**Type:** character vector

**Value:** `'1' | '<integer>'`

**Default:** `'1'`

**Argument name — Specify output argument name**

`u` (default) | character vector

Specify output argument name for the function prototype displayed on the face of the Simulink Function block.

`u`

Default name of the output argument.

character vector

Name of the output argument.

**Programmatic Use**

**Block parameter:** ArgumentName

**Type:** character vector
**Value:** 'u' | '<character vector>'
**Default:** 'u'

**Minimum — Specify minimum value for block input**

[] (default) | number

Specify the minimum value for the block input signal.

**Note** If you specify a bus object as the data type for this block, do not set the minimum value for bus data on the block. Simulink ignores this setting. Instead, set the minimum values for bus elements of the bus object specified as the data type. For information on the Minimum property of a bus element, see Simulink.BusElement.

Simulink uses this value to perform Simulation range checking (see “Signal Ranges”) and automatic scaling of fixed-point data types.

[]

Minimum value not specified.

number

Finite real double scalar value.

**Programmatic Use**

**Block parameter:** OutMin

**Type:** character vector

**Value:** '[]' | '<number>'

**Default:** '[]'

**Maximum — Specify maximum value for block input**

[] (default) | number

Specify the maximum value for the block input signal.

**Note** If you specify a bus object as the data type for this block, do not set the maximum value for bus data on the block. Simulink ignores this setting. Instead, set the maximum values for bus elements of the bus object specified as the data type. For information on the Maximum property of a bus element, see Simulink.BusElement.
Simulink uses this value to perform Simulation range checking (see “Signal Ranges”) and automatic scaling of fixed-point data types.

[]
  Max mum value not specified.
number
  Finite real double scalar value.

**Programmatic Use**

**Block parameter**: OutMax
  **Type**: character vector
  **Value**: '[]' | '<number>'
  **Default**: '[]'

**Data type — Specify block input data type**

double (default) | single | int8 | uint8 | int16 | uint16 | int32 | uint32 | int64 | uint64 | boolean | fixdt(1,16) | fixdt(1,16,0) | fixdt(1,16,2^,0) | Enum:
  <class name> | Bus: <object name> | <data type expression>

Specify block input data type. For more information, see

- Outport
- “Specify Data Types Using Data Type Assistant”

double
  Data type is double.
single
  Data type is single.
int8
  Data type is int8.
uint8
  Data type is uint8.
int16
  Data type is int16.
uint16
  Data type is uint16.
int32
  Data type is int32.
uint32
  Data type is uint32.
int64
  Data type is int64.
uint64
  Data type is uint64.
boolean
  Data type is boolean.
fixdt(1,16,0)
  Data type is fixed point fixdt(1,16,0).
fixdt(1,16,2^0,0)
  Data type is fixed point fixdt(1,16,2^0,0).
Enum: <class name>
  Data type is enumerated, for example, Enum: Basic Colors.
Bus: <object name>
  Data type is a bus object.
<data type expression>
  The name of a data type object, for example Simulink.NumericType

Do not specify a bus object as the expression.

**Programmatic Use**

**Block parameter:** OutDataTypeStr

**Type:** character vector

**Value:** 'double' | 'single' | 'int8' | 'uint8' | 'int16' | 'uint16' | 'int32' | 'uint32' | 'int64' | 'uint64' | 'boolean' | '<fixdt(1,16)' | 'fixdt(1,16,0)' | 'fixdt(1,16,2^0,0)' | '<data type expression>'

**Default:** 'double'

**Lock output data type setting against changes by the fixed-point tools** — Prevent fixed-point tools from overriding data type

off (default) | on
Control changes to data type settings from the Fixed-Point Tool and the Fixed-Point Advisor. For more information, see “Use Lock Output Data Type Setting” (Fixed-Point Designer).

- **On**
  - Locks all data type settings for this block.

- **Off**
  - Allows the Fixed-Point Tool and the Fixed-Point Advisor to change data type settings for this block.

**Programmatic Use**

**Block parameter:** LockScale

- **Type:** character vector
- **Value:** `'off'` | `'on'`
- **Default:** `'off'`

**Port dimensions — Specify port dimensions**

`1` (default) | `n` | `[m n]`

Specify the dimensions of the output argument signal from the block. For more information, see Outport.

- `1`
  - Inherit port dimensions.

- `n`
  - Vector signal of width `n`.

- `[m n]`
  - Matrix signal having `m` rows and `n` columns.

**Programmatic Use**

**Block parameter:** PortDimensions

- **Type:** character vector
- **Value:** `'1'` | `'n'` | `[m n]`
- **Default:** `'1'`

**Signal type — Select real or complex signal**

- **real (default)** | **complex**

Select real or complex signal. For more information, see Outport.
real
    Specify the signal type as a real number.
complex
    Specify the signal type as a complex number.

**Programmatic Use**
**Block parameter:** SignalType  
**Type:** character vector  
**Value:** 'real' | 'complex'
**Default:** 'real'

**See Also**
Argument Inport | Function Caller | Simulink Function

**Topics**
“Simulink Functions Overview”
“Argument Specification for Simulink Function Blocks”

**Introduced in R2014b**
**ASCII to String**

Uint8 vector signal to string signal  
**Library:** Simulink / String

**Description**

The ASCII to String block converts uint8 vector signals to string signals. The block treats each element in the input vector as an ASCII value during the conversion. For example, the block converts an input vector of \([72 \ 101 \ 108 \ 108 \ 111]\) to the string "Hello".

**Ports**

**Input**

**Port_1 — ASCII signal**

*vector*

ASCII signal, specified as a vector.

While using dynamic strings, if the length of the input vector exceeds the number of characters specified in the configuration parameter **Buffer size of dynamically-sized string (bytes)** (256 by default), the ASCII to String block truncates the string output to the buffer size-1 (for example, 255), for generated code. To avoid truncation, increase the value of the **Buffer size of dynamically-sized string (bytes)** configuration parameter.

Example: \([088 \ 099]\)

Data Types: uint8

**Output**

**Port_1 — Converted string signal**

*scalar*
Converted string signal from input ASCII signal, specified as a scalar. The block converts each ASCII element in the vector into its alphanumeric equivalent and outputs all elements concatenated into one string.

Data Types: string

## Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>integer</th>
<th>string</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
</tr>
</tbody>
</table>

## Extended Capabilities

### C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

### See Also
- Compose String
- Scan String
- String Compare
- String Concatenate
- String Constant
- String Find
- String Length
- String To ASCII
- String To Enum
- String to Double
- String to Single
- Substring
- To String

### Topics
- “String Data Type Conversions”
- “Simulink Strings”
Introduced in R2018a
**Assertion**

Check whether signal is zero  
**Library:** Simulink / Model Verification  
HDL Coder / Model Verification

**Description**

The Assertion block checks whether any of the elements of the input signal are zero. If all elements are nonzero, the block does nothing. If any element is zero, the block halts the simulation, by default, and displays an error message. Use the block parameter dialog box to:

- Specify that the block displays a warning message when the assertion fails but allows the simulation to continue.
- Specify a MATLAB expression to evaluate when the assertion fails.
- Enable or disable the assertion.

Use the blocks in the Model Verification library to facilitate creation of self-validating models. For example, you can use model verification blocks to test that signals do not exceed specified limits during simulation. When you are satisfied that a model is correct, you can turn error checking off by disabling the verification blocks. You do not have to remove them from the model. If you need to modify a model, you can temporarily turn the verification blocks back on to ensure that your changes do not break the model.

**Ports**

**Input**

*Port_1 — Input signal*

scalar | vector | matrix
Input signal to the assertion check.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated

Parameters

Enable assertion — Enable or disable the check

on (default) | off

Clearing this check box disables the block and causes the model to behave as if the block does not exist. You can set the Model Verification block enabling setting in the Configuration Parameters to enable or disable all model verification blocks in a model regardless of the setting of this option.

Command-Line Information
Parameter: enabled
Type: character vector
Values: 'on' | 'off'
Default: 'on'

Simulation callback when assertion fails (optional) — Expression to evaluate when assertion fails

' ' (default) | MATLAB expression

Specify a MATLAB expression to evaluate when the assertion fails. Because the expression is evaluated in the MATLAB workspace, define all variables used in the expression in that workspace.

Command-Line Information
Parameter: callback
Type: character vector
Values: MATLAB expression
Default: ''

Stop simulation when assertion fails — Halt simulation when check fails

on (default) | off

Select this check box to indicate that the block halts simulation when the check fails. Clear to indicate that the software displays a warning and continues the simulation.
Command-Line Information
Parameter: stopWhenAssertionFail
Type: character vector
Values: 'on' | 'off'
Default: 'on'

Sample time — Specify sample time as a value other than -1
-1 (default) | scalar | vector

Specify the sample time as a value other than -1. For more information, see “Specify Sample Time”.

Dependencies
This parameter is not visible unless it is explicitly set to a value other than -1. To learn more, see “Blocks for Which Sample Time Is Not Recommended”.

Programmatic Use
Block Parameter: SampleTime
Type: character vector
Values: scalar or vector
Default: '-1'

Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>double</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

For information about how Simulink Coder™ generated code handles Model Verification blocks, see “Debug” (Simulink Coder).

Not recommended for production code.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture
This block can be used for checking whether the input signal is zero during simulation, but is not included in the hardware implementation.

PLC Code Generation
Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

See Also
Check Dynamic Lower Bound | Check Dynamic Upper Bound | Check Static Lower Bound | Check Static Upper Bound

Introduced before R2006a
Assignment

Assign values to specified elements of signal

Library: Simulink / Math Operations
        HDL Coder / Math Operations

Description

The Assignment block assigns values to specified elements of the signal. You specify the indices of the elements to be assigned values either by entering the indices in the block dialog box or by connecting an external indices source or sources to the block. The signal at the block data port, \( U \), specifies values to be assigned to \( Y \). The block replaces the specified elements of \( Y \) with elements from the data signal.

Based on the value you enter for the Number of output dimensions parameter, a table of index options is displayed. Each row of the table corresponds to one of the output dimensions in Number of output dimensions. For each dimension, you can define the elements of the signal to work with. Specify a vector signal as a 1-D signal and a matrix signal as a 2-D signal. To enable an external index port, in the corresponding row of the table, set Index Option to Index vector (port).

For example, assume a 5-D signal with a one-based index mode. The table in the Assignment block dialog changes to include one row for each dimension. If you define each dimension with the following entries:

<table>
<thead>
<tr>
<th>Row</th>
<th>Index Option</th>
<th>Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Assign all</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Index vector (dialog)</td>
<td>[1 3 5]</td>
</tr>
<tr>
<td>3</td>
<td>Starting index (dialog)</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>Starting index (port)</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Index vector (port)</td>
<td></td>
</tr>
</tbody>
</table>
The assigned values are \( Y(1:end,[1\ 3\ 5],4:3+\text{size}(U,3),\text{Idx4}:\text{Idx4}+\text{size}(U,4)-1,\text{Idx5})=U \), where \( \text{Idx4} \) and \( \text{Idx5} \) are the input ports for dimensions 4 and 5.

When using the Assignment block in normal mode, Simulink initializes block outputs to zero even if the model does not explicitly initialize them. In accelerator mode, Simulink converts the model into an S-Function. This involves code generation. The code generated may not do implicit initialization of block outputs. In such cases, you must explicitly initialize the model outputs.

You can use the block to assign values to vector, matrix, or multidimensional signals.

You can use an array of buses as an input signal to an Assignment block.

**Assignment Block in Conditional Subsystem**

If you place an Assignment block in a conditional subsystem block, a signal buffer can be inserted in many cases, and merging of signals from Assignment blocks with partial writes can cause an error.

However, if you select the **Ensure outport is virtual** check box for the conditional subsystem Outport block, such cases are supported and partial writes to arrays using Assignment blocks are possible.
Ports

Input

Y₀ — Input initialization signal
scalar | vector

The initialization signal for the output signal. If an element is not assigned another value, then the value of the output element matches this input signal value.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point | Boolean | enumerated | bus

U — Input data port
scalar | vector

Value assigned to the output element when specified.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point | Boolean | enumerated | bus

IndexN — Nth index signal
scalar | vector

External port specifying an index for the assignment of the corresponding output element.

Dependencies

To enable an external index port, in the corresponding row of the Index Option table, set Index Option to Index vector (port) or Starting index (port).

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point | enumerated

Output

Y — Output signal with assigned values
scalar | vector

The output signal with assigned values for the specified elements.
Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point | enumerated | bus

Parameters

**Number of output dimensions — Number of dimensions of the output signal**
1 (default) | integer

Enter the number of dimensions of the output signal.

**Command-Line Information**
Parameter: NumberOfDimensions
Type: character vector
Values: integer
Default: '1'

**Index mode — Index mode**
One-based (default) | Zero-based

Select the indexing mode. If One-based is selected, an index of 1 specifies the first element of the input vector. If Zero-based is selected, an index of 0 specifies the first element of the input vector.

**Command-Line Information**
Parameter: IndexMode
Type: character vector
Values: 'Zero-based' | 'One-based'
Default: 'One-based'

**Index Option — Index method for elements**
Index vector (dialog) (default) | Assign all | Index vector (port) | Starting index (dialog) | Starting index (port)

Define, by dimension, how the elements of the signal are to be indexed. From the list, select:

<table>
<thead>
<tr>
<th>Menu Item</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assign all</td>
<td>This is the default. All elements are assigned.</td>
</tr>
<tr>
<td>Menu Item</td>
<td>Action</td>
</tr>
<tr>
<td>------------------------</td>
<td>-------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Index vector (dialog)</td>
<td>Enables the Index column. Enter the indices of elements.</td>
</tr>
<tr>
<td>Index vector (port)</td>
<td>Disables the Index column. The index port defines the indices of elements.</td>
</tr>
<tr>
<td>Starting index (dialog)</td>
<td>Enables the Index column. Enter the starting index of the range of elements to be assigned values.</td>
</tr>
<tr>
<td>Starting index (port)</td>
<td>Disables the Index column. The index port defines the starting index of the range of elements to be assigned values.</td>
</tr>
</tbody>
</table>

If you choose Index vector (port) or Starting index (port) for any dimension in the table, you can specify one of these values for the Initialize output (Y) parameter:

- Initialize using input port <Y0>
- Specify size for each dimension in table

Otherwise, Y0 always initializes output port Y.

The Index and Output Size columns are displayed as relevant.

**Command-Line Information**

**Parameter:** IndexOptionArray  
**Type:** character vector  
**Values:** 'Assign all' | 'Index vector (dialog)' | 'Index option (port)' | 'Starting index (dialog)' | 'Starting index (port)'  
**Default:** 'Index vector (dialog)'

**Index — Index of elements**

1 (default) | integer

If the Index Option is Index vector (dialog), enter the index of each element you are interested in.

If the Index Option is Starting index (dialog), enter the starting index of the range of elements to be selected. The number of elements from the starting point is determined by the size of this dimension at U.
Command-Line Information
Parameter: IndexParamArray
Type: character vector
Values: cell array
Default: '{ }'

Output Size — Width of the block output signal
1 (default) | integer
Enter the width of the block output signal.

Dependencies
To enable this column, select Specify size for each dimension in table for the Initialize output (Y) parameter.

Command-Line Information
Parameter: OutputSizeArray
Type: character vector
Values: cell array
Default: '{ }'

Initialize output (Y) — How to initialize the output signal
Initialize using input port <Y0> (default) | Specify size for each dimension in the table
Specify how to initialize the output signal.

- Initialize using input port <Y0> - Signal at the input port Y0 initializes the output.
- Specify size for each dimension in table - Requires you to specify the width of the block's output signal in the Output Size parameter. If the output has unassigned elements, the value of those elements is undefined.

Dependency
Enabled when you set Index Option to Index vector (port) or Starting index (port).

Command-Line Information
Parameter: OutputInitialize
Type: character vector
Values: 'Initialize using input port <Y0>' | 'Specify size for each dimension in table'
Default: 'Initialize using input port <Y0>'

Action if any output element is not assigned — Specify whether to produce a warning or error if you have not assigned all output elements
Error (default) | Warning | None

Specify whether to produce a warning or error if you have not assigned all output elements. Options include:

- Error — Simulink software terminates the simulation and displays an error.
- Warning — Simulink software displays a warning and continues the simulation.
- None — Simulink software takes no action.

Command-Line Information
Parameter: DiagnosticForDimensions
Type: character vector
Values: 'Error' | 'Warning' | 'None'
Default: 'None'

Sample time — Specify sample time as a value other than -1
-1 (default) | scalar | vector

Specify the sample time as a value other than -1. For more information, see “Specify Sample Time”.

Dependencies
This parameter is not visible unless it is explicitly set to a value other than -1. To learn more, see “Blocks for Which Sample Time Is Not Recommended”.

Programmatic Use
Block Parameter: SampleTime
Type: character vector
Values: scalar or vector
Default: '-1'
Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>double</th>
<th>enumerated</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture
This block has a single, default HDL architecture.
HDL Block Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConstrainedOutputPipeline</td>
<td>Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. For more details, see “ConstrainedOutputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td>InputPipeline</td>
<td>Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “InputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td>OutputPipeline</td>
<td>Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “OutputPipeline” (HDL Coder).</td>
</tr>
</tbody>
</table>

Complex Data Support

This block supports code generation for complex signals.

Restrictions

- 3-dimensional matrix inputs are not supported. You can use 1-D vectors and 2-D matrices with the block.
- Variable-size signals are not supported for code generation.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

Design and simulate fixed-point systems using Fixed-Point Designer™.

See Also

“Combine Buses into an Array of Buses” | Bus Assignment
Introduced before R2006a
Backlash

Model behavior of system with play

Library:  
Simulink / Discontinuities  
HDL Coder / Discontinuities

Description

The Backlash block implements a system in which a change in input causes an equal change in output, except when the input changes direction. When the input changes direction, the initial change in input has no effect on the output. The amount of side-to-side play in the system is referred to as the deadband. The deadband is centered about the output. This figure shows an initial state, with the default deadband width of 1 and initial output of 0.

A system with play can be in one of three modes.

<table>
<thead>
<tr>
<th>Mode</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Disengaged</td>
<td>Inside deadband zone.</td>
<td>Remains constant.</td>
</tr>
<tr>
<td>Engaged-positive direction</td>
<td>Outside deadband zone and increasing.</td>
<td>Equals input minus half of deadband width.</td>
</tr>
<tr>
<td>Engaged-negative direction</td>
<td>Outside deadband zone and decreasing.</td>
<td>Equals input plus half of deadband width.</td>
</tr>
</tbody>
</table>

The Initial output parameter value defines the initial center of the deadband zone.

This table shows output values when initial conditions are: Deadband width = 2 and Initial output = 5.
<table>
<thead>
<tr>
<th>Output Value</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>4 &lt; input &lt; 6</td>
</tr>
<tr>
<td>input + 1</td>
<td>input &lt; 4</td>
</tr>
<tr>
<td>input - 1</td>
<td>input &gt; 6</td>
</tr>
</tbody>
</table>

For example, you can use the Backlash block to model the meshing of two gears. The input and output are both shafts with a gear on one end, and the input shaft drives the output shaft. Extra space between the gear teeth introduces play. The width of this spacing is the **Deadband width** parameter. If the system is disengaged initially, the **Initial output** parameter defines the output.

These figures illustrate operation when the initial input is within the deadband and the system begins in disengaged mode.

When the input increases and reaches the end of the deadband, it engages the output. The output remains at its previous value.

After the input engages the output, the output changes by the same amount as the input.

If the input reverses direction, it disengages from the output. The output remains constant until the input reaches the end of the deadband and engages again.
Ports

Input

Port_1 — Input signal
scalar | vector

Input signal to the backlash algorithm. The value of this signal is either in the deadband or engaging the output in a positive or negative direction.

Data Types: double | single | int8 | int16 | int32 | uint8 | uint16 | uint32

Output

Port_1 — Output signal
scalar | vector

Output signal after the backlash algorithm is applied to the input signal. When the input is in the deadband, then the output remains unchanged. If the input is engaged with the output, then the output changes an equal amount as the input.

Data Types: double | single | int8 | int16 | int32 | uint8 | uint16 | uint32

Parameters

Deadband width — Specify the width of the deadband
1 (default) | scalar | vector

Specify the size of the deadband zone centered on the output value. When the input signal is inside the deadband, then a change in input does not cause a change in output. When the input signal is outside of the deadband, then the output changes an equal amount as the input.

Programmatic Use

Block Parameter: BacklashWidth
Type: character vector
Values: real scalar or vector
Default: '1'
**Initial output — Specify the initial output value**

0 (default) | scalar | vector

Specify the initial center of the deadband zone. If the initial input value is in the deadband zone, then the output value is equal to **Initial output**. If the initial input value is outside of the deadband zone, then the output value is **Initial output** plus or minus half of the deadzone width.

**Programmatic Use**

**Block Parameter:** InitialOutput  
**Type:** character vector  
**Values:** real scalar or vector  
**Default:** '0'

**Input processing — Specify sample- or frame-based processing**

Elements as channels (sample based) (default) | Columns as channels (frame based)

Specify whether the block performs sample- or frame-based processing:

- **Columns as channels (frame based)** — Treat each column of the input as a separate channel (frame-based processing).

**Note** Frame-based processing requires a DSP System Toolbox™ license.

For more information, see “Sample- and Frame-Based Concepts” (DSP System Toolbox).

- **Elements as channels (sample based)** — Treat each element of the input as a separate channel (sample-based processing).

Use **Input processing** to specify whether the block performs sample- or frame-based processing. The block accepts frame-based signals for the input u. All other input signals must be sample-based.

<table>
<thead>
<tr>
<th>Input Signal u</th>
<th>Input Processing Mode</th>
<th>Block Works?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sample based</td>
<td>Sample based</td>
<td>Yes</td>
</tr>
<tr>
<td>Frame based</td>
<td></td>
<td>No, produces an error</td>
</tr>
<tr>
<td>Sample based</td>
<td>Frame based</td>
<td>Yes</td>
</tr>
<tr>
<td>Frame based</td>
<td></td>
<td>Yes</td>
</tr>
</tbody>
</table>
For more information about these two processing modes, see “Sample- and Frame-Based Concepts” (DSP System Toolbox).

**Programmatic Use**

**Block Parameter:** InputProcessing  
**Type:** character vector  
**Values:** 'Columns as channels (frame based)' | 'Elements as channels (sample based)'  
**Default:** 'Elements as channels (sample based)'

**Enable zero-crossing detection — Enable zero-crossing detection**

on (default) | Boolean

Select to enable zero-crossing detection. For more information, see “Zero-Crossing Detection” in the Simulink documentation.

**Programmatic Use**

**Block Parameter:** ZeroCross  
**Type:** character vector, string  
**Values:** 'off' | 'on'  
**Default:** 'on'

**Block Characteristics**

<table>
<thead>
<tr>
<th>Data Types</th>
<th>double</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>no</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>yes</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Extended Capabilities

**C/C++ Code Generation**
Generate C and C++ code using Simulink® Coder™.

Generated code relies on `memcpy` or `memset` function (`string.h`) in certain conditions.

**HDL Code Generation**
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

**HDL Architecture**
This block has a single, default HDL architecture.

**HDL Block Properties**

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConstrainedOutput Pipeline</td>
<td>Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. For more details, see “ConstrainedOutputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td>InputPipeline</td>
<td>Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “InputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td>OutputPipeline</td>
<td>Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “OutputPipeline” (HDL Coder).</td>
</tr>
</tbody>
</table>

**Restrictions**

The **Deadband width** and **Initial output** parameters support only scalar values.
See Also
Dead Zone

Introduced before R2006a
Unresolved Link

Indicate unresolved reference to library block

"mylibrary" not found

Description

This block indicates an unresolved reference to a library block (see “Linked Blocks”). You can use this block's parameter dialog box to fix the reference to point to the actual location of the library block.

Parameters

Details

The Details field contains a description of the cause of the unresolved link. Simulink tries to help you find and install missing products that a model needs to run. For missing products, the block description provides a link. Click the link to open Add-On Explorer and install the missing products.
You can customize the Unresolved Link block description for your library to include URLs as follows:

```matlab
set_param(library1,'libraryinfo','https://www.mathworks.com');
```
Here, `library1` is the name of the library for which you want to change the description, and `libraryinfo` is the property that provides the description of the unresolved link.

**Source block**

Path of the library block that this link represents. To fix a bad link, either click the link in the description to open Add-On Explorer and install a missing product, or edit the **Source block** field to the correct path of the library block. Then select Apply or OK to apply the fix and close the dialog box.

Alternatively, to fix an unresolved link, you can:

- Delete the unresolved block and copy the library block back into your model.
- Add the folder that contains the required library to the MATLAB path and then, in the **Modeling** tab, click **Update Model**.

**Source type**

Type of library block that this link represents.

**See Also**

**Topics**

“Linked Blocks”

“Fix Unresolved Library Links”

**Introduced in R2014a**
Band-Limited White Noise

Introduce white noise into continuous system

Library: Simulink / Sources

Description

The Band-Limited White Noise block generates normally distributed random numbers that are suitable for use in continuous or hybrid systems.

Simulation of White Noise

Theoretically, continuous white noise has a correlation time of 0, a flat power spectral density (PSD), and a total energy of infinity. In practice, physical systems are never disturbed by white noise, although white noise is a useful theoretical approximation when the noise disturbance has a correlation time that is very small relative to the natural bandwidth of the system.

In Simulink software, you can simulate the effect of white noise by using a random sequence with a correlation time much smaller than the shortest time constant of the system. The Band-Limited White Noise block produces such a sequence. The correlation time of the noise is the sample rate of the block. For accurate simulations, use a correlation time much smaller than the fastest dynamics of the system. You can get good results by specifying

\[ tc \approx \frac{1}{100} \frac{2\pi}{f_{\text{max}}}, \]

where \( f_{\text{max}} \) is the bandwidth of the system in rad/sec.
Comparison with the Random Number Block

The primary difference between this block and the Random Number block is that the Band-Limited White Noise block produces output at a specific sample rate. This rate is related to the correlation time of the noise.

Usage with the Averaging Power Spectral Density Block

The Band-Limited White Noise block specifies a two-sided spectrum, where the units are Hz. The Averaging Power Spectral Density block specifies a one-sided spectrum, where the units are the square of the magnitude per unit radial frequency: \( \text{mag}^2/\text{rad/sec} \). When you feed the output of a Band-Limited White Noise block into an Averaging Power Spectral Density block, the average PSD value is \( \pi \) times smaller than the Noise power of the Band-Limited White Noise block. This difference is the result of converting the units of one block to the units of the other, \( 1/(1/2)(2\pi) = \frac{1}{\pi} \), where:

- \( 1/2 \) is the factor for converting from a two-sided to one-sided spectrum.
- \( 2\pi \) is the factor for converting from Hz to rad/sec.

Ports

Output

Port_1 — Normally distributed random numbers
scalar | vector | matrix | N-D array

Normally distributed random numbers specified as a scalar, vector, matrix, or N-D array.

Data Types: double

Parameters

Noise power — Height of PSD of white noise
[0.1] (default) | scalar | vector | matrix | N-D array

Specify the height of the PSD of the white noise as a scalar, vector, matrix, or N-D array of positive values.
**Programmatic Use**

**Block Parameter:** Cov  
**Type:** character vector  
**Values:** scalar | vector | matrix | N-D array  
**Default:** 

'\[0.1\]'

**Sample time — Correlation time of noise**

0.1 (default) | scalar | vector

Correlation time of the noise. For more information, see “Specify Sample Time”.

**Programmatic Use**

**Block Parameter:** Ts  
**Type:** character vector  
**Values:** scalar | vector  
**Default:** '0.1'

**Seed — Starting seed**

[23341] (default) | scalar | vector | matrix | N-D array

Specify the starting seed for the random number generator as a scalar, vector, matrix, or N-D array. Values must be positive, real-valued, and finite.

**Programmatic Use**

**Block Parameter:** seed  
**Type:** character vector  
**Values:** scalar | vector | matrix | N-D array  
**Default:** '23341'

**Interpret vector parameters as 1-D — Treat vector parameters as 1-D**

on (default) | off

Select to output a 1-D array when the block parameters are vectors. Otherwise, output a 2-D array one of whose dimensions is 1. For more information, see “Determining the Output Dimensions of Source Blocks”.

**Programmatic Use**

**Block Parameter:** VectorParams1D  
**Type:** character vector  
**Values:** 'on' | 'off'

**Default:** 'on'
Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>double</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>no</td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
</tr>
</tbody>
</table>

Algorithms

To produce the correct intensity of this noise, the covariance of the noise is scaled to reflect the implicit conversion from a continuous PSD to a discrete noise covariance. The appropriate scale factor is \( \frac{1}{tc} \), where \( tc \) is the correlation time of the noise. This scaling ensures that the response of a continuous system to the approximate white noise has the same covariance as the system would have to true white noise. Because of this scaling, the covariance of the signal from the Band-Limited White Noise block is not the same as the **Noise power** (intensity) parameter. This parameter is actually the height of the PSD of the white noise. This block approximates the covariance of white noise as the **Noise power** divided by \( tc \).

Extended Capabilities

**C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

Cannot use inside a triggered subsystem hierarchy.
See Also
Random Number

Topics
“Sample Time”

Introduced before R2006a
Bias

Add bias to input

Library:
- Simulink / Math Operations
- HDL Coder / HDL Floating Point Operations
- HDL Coder / Math Operations

Description

The Bias block adds a bias, or offset, to the input signal according to

\[ Y = U + \text{bias} \]

where \( U \) is the block input and \( Y \) is the output.

Ports

Input

Port_1 — Input signal
scalar | vector

Input signal to which the bias is added to create the output signal.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

Output

Port_1 — Output signal
scalar | vector

Output signal resulting from adding the bias to the input signal.
Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

### Parameters

**Bias — Offset to add to the input signal**

0.0 (default) | scalar | vector

Specify the value of the offset to add to the input signal.

**Programmatic Use**

**Block Parameter:** Bias

**Type:** character vector

**Values:** real, finite

**Default:** '0.0'

**Saturate on integer overflow — Choose the behavior when integer overflow occurs**

on (default) | boolean

<table>
<thead>
<tr>
<th>Action</th>
<th>Reasons for Taking This Action</th>
<th>What Happens for Overflows</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Select this check box.</td>
<td>Your model has possible overflow, and you want explicit saturation protection in the generated code.</td>
<td>Overflows saturate to either the minimum or maximum value that the data type can represent.</td>
<td>The maximum value that the int8 (signed, 8-bit integer) data type can represent is 127. Any block operation result greater than this maximum value causes overflow of the 8-bit integer. With the check box selected, the block output saturates at 127. Similarly, the block output saturates at a minimum output value of -128.</td>
</tr>
<tr>
<td>Action</td>
<td>Reasons for Taking This Action</td>
<td>What Happens for Overflows</td>
<td>Example</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>------------------------------------------------------------------------------------------------</td>
<td>-------------------------------------------------------------------------------------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Do not select this check box.</td>
<td>You want to optimize efficiency of your generated code. You want to avoid overspecifying how a block handles out-of-range signals. For more information, see “Check for Signal Range Errors”.</td>
<td>Overflows wrap to the appropriate value that is representable by the data type.</td>
<td>The maximum value that the int8 (signed, 8-bit integer) data type can represent is 127. Any block operation result greater than this maximum value causes overflow of the 8-bit integer. With the check box cleared, the software interprets the overflow-causing value as int8, which can produce an unintended result. For example, a block result of 130 (binary 1000 0010) expressed as int8, is -126.</td>
</tr>
</tbody>
</table>

When you select this check box, saturation applies to every internal operation on the block, not just the output or result. Usually, the code generation process can detect when overflow is not possible. In this case, the code generator does not produce saturation code.

**Programmatic Use**  
**Block Parameter:** SaturateOnIntegerOverflow  
**Type:** character vector  
**Value:** 'off' | 'on'  
**Default:** 'on'

## Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>double</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture
This block has a single, default HDL architecture.

HDL Block Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multidimensional Signals</td>
<td>no</td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>yes</td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
</tr>
<tr>
<td>ConstrainedOutputPipeline</td>
<td>Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. For more details, see “ConstrainedOutputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td>InputPipeline</td>
<td>Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “InputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td>OutputPipeline</td>
<td>Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “OutputPipeline” (HDL Coder).</td>
</tr>
</tbody>
</table>

**PLC Code Generation**  
Generate Structured Text code using Simulink® PLC Coder™.

**Fixed-Point Conversion**  
Design and simulate fixed-point systems using Fixed-Point Designer™.

**See Also**  
Add | Divide | Gain

**Introduced before R2006a**
Bit Clear

Set specified bit of stored integer to zero

**Library:**
- Simulink / Logic and Bit Operations
- HDL Coder / Logic and Bit Operations

**Description**

The Bit Clear block sets the specified bit, given by its index, of the stored integer to zero. Scaling is ignored.

You can specify the bit to be set to zero with the **Index of bit** parameter, where bit zero is the least significant bit.

**Ports**

The Bit Clear block supports Simulink integer, fixed-point, and Boolean data types. The block does not support true floating-point data types or enumerated data types.

**Input**

*Port_1 — Input signal*

scalar or vector

The input signal is the specified bit of the stored integer.

Data Types: single | double | Boolean | fixed point

**Output**

*Port_1 — Output signal*

scalar or vector
The output consists of the specified bit set to zero.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus

**Parameters**

**Index of bit — Index of bit**
0 (default) | scalar or vector

Index of bit where bit 0 is the least significant bit.

**Programmatic Use**
*Block Parameter:* iBit
*Type:* scalar or vector
*Values:* {'0'}
*Default:* '0'

**Block Characteristics**

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>fixed point</th>
<th>integer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

a. Bit operations are not recommended for use with Boolean signals.
Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture
This block has a single, default HDL architecture.

HDL Block Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConstrainedOutputPipeline</td>
<td>Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. For more details, see “ConstrainedOutputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td>InputPipeline</td>
<td>Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “InputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td>OutputPipeline</td>
<td>Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “OutputPipeline” (HDL Coder).</td>
</tr>
</tbody>
</table>

PLC Code Generation
Generate Structured Text code using Simulink® PLC Coder™.
Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

See Also
Bit Rotate | Bit Set | Bit Shift | Bitwise Operator

Introduced before R2006a
Bit Set

Set specified bit of stored integer to one

Library: Simulink / Logic and Bit Operations
        HDL Coder / Logic and Bit Operations

Description

The Bit Set block sets the specified bit of the stored integer to one. Scaling is ignored.

You can specify the bit to be set to one with the Index of bit parameter, where bit zero is the least significant bit.

Ports

Input

Port_1 — Input signal
scalar or vector

Input signal with the specified bit of the stored integer.

Data Types: single | double | Boolean | fixed point

Output

Port_1 — Output signal
scalar or vector

Output signal with the specified bit set to 1.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus
Parameters

Index of bit — Index of bit
0 (default) | scalar or vector

Index of bit where bit 0 is the least significant bit.

Programmatic Use
Block Parameter: iBit
Type: character vector
Values: positive integer
Default: '0'

Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean(^a)</th>
<th>fixed point</th>
<th>integer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\(^a\) Bit operations are not recommended for use with Boolean signals.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.
HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture
This block has a single, default HDL architecture.

HDL Block Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConstrainedOutputPipeline</td>
<td>Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. For more details, see “ConstrainedOutputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td>InputPipeline</td>
<td>Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “InputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td>OutputPipeline</td>
<td>Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “OutputPipeline” (HDL Coder).</td>
</tr>
</tbody>
</table>

PLC Code Generation
Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

See Also
Bit Clear | Bit Rotate | Bit Shift | Bitwise Operator
Introduced before R2006a
Bitwise Operator

Specified bitwise operation on inputs

Library: Simulink / Logic and Bit Operations
        HDL Coder / Logic and Bit Operations

Description

The Bitwise Operator block performs the bitwise operation that you specify on one or more operands. Unlike logic operations of the Logical Operator block, bitwise operations treat the operands as a vector of bits rather than a single value.

Restrictions on Block Operations

The Bitwise Operator block does not support shift operations. For shift operations, use the Shift Arithmetic block.

When configured as a multi-input XOR gate, this block performs modulo-2 addition according to the IEEE® Standard for Logic Elements.

Ports

Input

Port_1 — Input signal

scalar | vector

Input signal, specified as a scalar or vector.

- The NOT operator accepts only one input, which can be a scalar or a vector. If the input is a vector, the output is a vector of the same size containing the bitwise logical complements of the input vector elements.
• For a single vector input, the block applies the operation (except the NOT operator) to all elements of the vector.
• For two or more inputs, the block performs the operation between all of the inputs. If the inputs are vectors, the block performs the operation between corresponding elements of the vectors to produce a vector output.

Data Types: int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point

Output

Port_1 — Output signal
scalar | vector

The output signal specified as the output data type, which the block inherits from the driving block, must represent zero exactly. Data types that satisfy this condition include signed and unsigned integer data types.

The size of the block output depends on the number of inputs, the vector size, and the operator you select. If you do not specify a bit mask, the output is a scalar. If you do specify a bit mask, the output is a vector.

Data Types: int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point

Parameters

Operator — Bitwise logical operator
AND (default) | OR | NOR | NAND | XOR | NOT

Specify the bitwise logical operator for the block operands.

You can select one of these bitwise operations:

<table>
<thead>
<tr>
<th>Bitwise Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND</td>
<td>TRUE if the corresponding bits are all TRUE</td>
</tr>
<tr>
<td>OR</td>
<td>TRUE if at least one of the corresponding bits is TRUE</td>
</tr>
<tr>
<td>NAND</td>
<td>TRUE if at least one of the corresponding bits is FALSE</td>
</tr>
<tr>
<td>NOR</td>
<td>TRUE if no corresponding bits are TRUE</td>
</tr>
</tbody>
</table>
### Bitwise Operation

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XOR</td>
<td>TRUE if an odd number of corresponding bits are TRUE</td>
</tr>
<tr>
<td>NOT</td>
<td>TRUE if the input is FALSE (available only for single input)</td>
</tr>
</tbody>
</table>

**Programmatic Use**

**Block Parameter:** `logicop`

**Type:** character vector

**Values:** `'AND'` | `'OR'` | `'NAND'` | `'NOR'` | `'XOR'` | `'NOT'`

**Default:** `'AND'`

**Use bit mask — Select to use bit mask**

checked (default) | unchecked

Select to use the bit mask. Clearing this check box enables **Number of input ports** and disables **Bit Mask** and **Treat mask as**.

**Programmatic Use**

**Block Parameter:** `UseBitMask`

**Type:** character vector

**Values:** `'off'` | `'on'`

**Default:** `'on'`

**Number of input ports — Number of input signals**

1 (default) | integer

Specify the number of inputs. You can have more than one input ports.

**Dependency**

Clearing the **Use bit mask** check box enables **Number of input ports** and disables **Bit Mask** and **Treat mask as**.

**Programmatic Use**

**Block Parameter:** `NumInputPorts`

**Type:** character vector

**Values:** positive integer

**Default:** `'1'`

**Bit Mask — Bit mask to associate with a single input**

bin2dec (default)

Specify the bit mask to associate with a single input.
You can use the bit mask to set or clear a bit on the input.

<table>
<thead>
<tr>
<th>To perform a...</th>
<th>Set the Operator parameter to...</th>
<th>And create a bit mask with...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit set</td>
<td>OR</td>
<td>A 1 for each corresponding input bit that you want to set to 1</td>
</tr>
<tr>
<td>Bit clear</td>
<td>AND</td>
<td>A 0 for each corresponding input bit that you want to set to 0</td>
</tr>
</tbody>
</table>

Suppose you want to set the fourth bit of an 8-bit input vector. The bit mask would be 00010000, which you can specify as $2^4$ for the **Bit Mask** parameter. To clear the bit, the bit mask would be 11101111, which you can specify as $2^7+2^6+2^5+2^3+2^2+2^1+2^0$ for the **Bit Mask** parameter.

**Tip** Do not use a mask greater than 53 bits. Otherwise, an error message appears during simulation.

**Dependency**

This parameter is available only when you select **Use bit mask**.

**Programmatic Use**

<table>
<thead>
<tr>
<th>Block Parameter: BitMask</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type: character vector</td>
</tr>
<tr>
<td>Values: positive integer</td>
</tr>
<tr>
<td>Default: 'bin2dec('11011001')'</td>
</tr>
</tbody>
</table>

**Treat mask as — Treat the mask as a real-world value or a stored integer**

Specify whether to treat the mask as a real-world value or a stored integer.

The encoding scheme is $V = SQ + B$, as described in “Scaling” (Fixed-Point Designer) in the Fixed-Point Designer™ documentation. **Real World Value** treats the mask as $V$. **Stored Integer** treats the mask as $Q$.

**Dependency**

This parameter is available only when you select **Use bit mask**.
Programmatic Use
Block Parameter: BitMaskRealWorld
Type: character vector
Values: 'Real World Value' | 'Stored Integer'
Default: 'Stored Integer'

Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean(^a)</th>
<th>fixed point</th>
<th>integer</th>
</tr>
</thead>
</table>

Direct Feedthrough
no

Multidimensional Signals
yes

Variable-Size Signals
no

Zero-Crossing Detection
no

---

a. Bit operations are not recommended for use with Boolean signals.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture
This block has a single, default HDL architecture.
HDL Block Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConstrainedOutput Pipeline</td>
<td>Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. For more details, see “ConstrainedOutputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td>InputPipeline</td>
<td>Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “InputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td>OutputPipeline</td>
<td>Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “OutputPipeline” (HDL Coder).</td>
</tr>
</tbody>
</table>

PLC Code Generation
Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

See Also
Compare To Constant | Compare To Zero | Logical Operator | Shift Arithmetic

Introduced before R2006a
Block Support Table

View data type support for Simulink blocks
Library: Simulink / Model-Wide Utilities

Description

The Block Support Table block helps you access a table that lists the data types that Simulink blocks support. To view the table, double-click the block.

Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>no</td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
</tr>
</tbody>
</table>

Tips

To open the Block Support Table from the command line, enter `showblockdatatypetable` at the MATLAB command prompt.
Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

The Block Support Table block is ignored during code generation.

See Also
showblockdatatypetable

Topics
“Data Types Supported by Simulink”

Introduced in R2007b
Bus Assignment

Replace specified bus elements

Library: Simulink / Signal Routing
          HDL Coder / Signal Routing

Description

The Bus Assignment block assigns the values of a signal to bus elements. Use a Bus Assignment block to change bus element values without adding Bus Selector and Bus Creator blocks that select bus elements and reassemble them into a bus.

Connect the bus signal to the first input port. To other input ports, connect one or more signals whose values you want to assign to a bus element. Use the Block Parameters dialog box to specify the bus elements to be replaced. The block displays an assignment input port for each such element. For an example of a model that uses a Bus Assignment block, see “Assign Signal Values to a Bus”.

By default, Simulink repairs broken selections in the Bus Assignment Block Parameters dialog boxes that are due to upstream bus hierarchy changes. Simulink generates a warning to highlight that it modified the model. To prevent Simulink from making these repairs automatically, in the Model Configuration Parameters > Diagnostics > Connectivity pane, set the “Repair bus selections” diagnostic to Error without repair.

Limitations

When using arrays of buses with a Bus Assignment block, these limitations apply:

- You can assign or replace a nested bus that is an array of buses. For a bus that is nested inside an array of buses, see “Assign into Arrays of Buses”.
- To replace a signal in an array of buses, use a Selector block to select the index for the bus element that you want to use with the Bus Assignment block. Then, use that selected bus element with the Bus Assignment block.
Ports

Input

Bus — Accept bus signal for bus element value assignment
real or complex values of any data type supported by Simulink

Input bus signals can have real or complex values of any data type supported by Simulink, including bus objects, arrays of buses, fixed-point, and enumerated data types. For details about data types, see Simulink, “Data Types Supported by Simulink”.

The signal connected to the assignment port must have the same structure, data type, and sample time as the bus element to which it corresponds. You can use a Rate Transition block to change the sample time of an individual signal or signals in a bus, to include the signal or bus in a nonvirtual bus.

:= — Accept signals whose value are assigned to bus elements
real or complex values of any data type supported by Simulink

Assignment input ports can accept signals can have real or complex values of any data type supported by Simulink, including bus objects, arrays of buses, fixed-point, and enumerated data types. You cannot use the Bus Assignment block to replace a bus that is nested within an array of buses. For details about data types, see Simulink, “Data Types Supported by Simulink”.

The Bus Assignment block assigns signals connected to its assignment input ports to specified elements of the bus connected to its bus input port. The block replaces the signals previously assigned to those elements. The change does not affect the composition of the bus; it affects only the values of the signals themselves. Signals not replaced are unaffected by the replacement of other signals.

Output

Bus — Output bus signal
virtual or nonvirtual bus

Bus that includes the assigned bus element values and the values of the bus elements of the input bus that you did not assign values to.
Parameters

Signals in the bus — Bus element signals of input bus
list of signal names

List of the bus element signals of the input bus signal. An arrow next to a signal name indicates that the input signal is a bus. To display the signals in an input bus, click the arrow.

Click any item in the list to select it. To find the source of the selected signal, click Find. Simulink opens and highlights the system containing the signal source. To move the currently selected signal into the adjacent list of signals to be assigned values (see Signals that are being assigned below), click Select. To refresh the display to reflect modifications to the bus connected to the block, click Refresh.

Filter by name — Filter set of displayed signals
text

Specify a search term to use for filtering a long list of input signals. Do not enclose the search term in quotation marks. The filter does a partial string search. To access filtering options, including using a regular expression for specifying the search term, click the button on the right of the Filter by name edit box.

Enable regular expression — Filter set of displayed input signals
off (default) | on

Enable the use of MATLAB regular expressions for filtering signal names. For example, entering t$ in the Filter by name edit box displays all signals whose names end with a lowercase t (and their immediate parents). For details, see “Regular Expressions” (MATLAB).

Dependencies

To access this parameter, click the button on the right of the Filter by name edit box.

Show filtered results as a flat list — Filter set of displayed input signals
off (default) | on

By default, the list displays as a tree list of filtered signals, based on the search text in the Filter by name edit box. To use a flat list format that uses dot notation to reflect the hierarchy of bus signals, select this parameter.
Dependencies

To access this parameter, click the button on the right of the Filter by name edit box.

Signals that are being assigned — Bus element signals to be assigned
list of signal names

Names of bus elements to be assigned values. This block displays an assignment input port for each bus element in this list. The label of the corresponding input port contains the name of the element. You can order the signals by using the Up, Down, or Remove. Port connectivity is maintained when you change the signal order.

If an input bus no longer contains a bus element, three question marks (???) appear before the name of that bus element. The reason for this event is that the bus has changed since the last time you refreshed the Bus Assignment block input and bus element assignment lists. To address this issue, either modify the bus to include a signal of the specified name or remove the name from the list of bus elements designated to be assigned values.

Programmatic Use

Block Parameter: OutputSignals
Type: character vector
Values: 'signal1' | 'signal2'
Default: none

Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>bus</th>
<th>double</th>
<th>enumerated</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
<th>string</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

Actual data type or capability support depends on block implementation.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

To learn more about using buses for HDL code generation, see “Buses” (HDL Coder) and “Use Bus Signals to Improve Readability of Model and Generate HDL Code” (HDL Coder).

HDL Architecture
This block has a single, default HDL architecture.

HDL Block Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConstrainedOutput</td>
<td>Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. For more details, see “ConstrainedOutputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td>Pipeline</td>
<td></td>
</tr>
<tr>
<td>InputPipeline</td>
<td>Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “InputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td>OutputPipeline</td>
<td>Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “OutputPipeline” (HDL Coder).</td>
</tr>
</tbody>
</table>
PLC Code Generation
Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

Actual data type or capability support depends on block implementation.

See Also
Bus Creator | Bus Selector

Topics
“Getting Started with Buses”
“Combine Buses into an Array of Buses”
“Composite Signals”
“Buses and Libraries”

Introduced before R2006a
**Bus Creator**

Create bus from input signals

**Library:**
- Simulink / Commonly Used Blocks
- Simulink / Signal Routing
- HDL Coder / Signal Routing

**Description**

The Bus Creator block combines a set of input signals into a bus. You can connect any signal types to the input ports, including other buses. You can access signals in a bus by using a Bus Selector block.

Elements of a bus must have unique names. If duplicate names are present, the Bus Creator block appends the port number to all input signal names. For signals that do not have names, the Bus Creator block generates names in the form `signaln`, where `n` is the port number connected to the signal. You can refer to signals by name when you search for their sources or select signals for connection to other blocks. For signal naming guidelines, see “Signal Names and Labels”.

**Tip** For models that include buses composed of many elements, consider using the In Bus Element and Out Bus Element blocks. You can use these bus element port blocks instead of Inport with Bus Selector blocks for inputs, and Outport with Bus Creator blocks for outputs.

Bus element port blocks:

- Reduce signal line complexity and clutter in a block diagram.
- Make it easier to change the interface incrementally.
- Allow access to a bus element closer to the point of usage, avoiding the use of a Bus Selector and Goto block configuration.
Ports

Input

**Port_1 — Accept signal to include in bus**
real or complex values of any data type supported by Simulink

Input signals can have real or complex values of any data type supported by Simulink, including bus objects, arrays of buses, fixed-point, and enumerated data types. For details about data types, see Simulink, “Data Types Supported by Simulink”.

Output

**Port_1 — Output bus signal**
virtual or nonvirtual bus

Bus that combines the input signals.

Parameters

**Number of inputs — Number of input ports**
2 (default) | integer

Number of inputs, not fewer than two. Increasing the number of connected ports adds empty ports below the connected ports. Before you simulate the model, make sure that an input signal is connected to each input port.

When you modify the **Number of inputs** parameter, click **Refresh** to update the list of signals.

**Tip** If all input ports are already connected, you can add an input port to the Bus Creator block by connecting another signal line to it. Interactively adding a port updates the **Number of inputs** parameter and adds the new signal to the list of signals in the bus.

Programmatic Use

**Block Parameter:** Inputs
**Type:** character vector
**Values:** integer greater than or equal to 2
**Default:** '2'

**Signals in the bus — Input signals**
list of signal names

List of input signals to combine into a bus. You can view all signals entering the block, including the signals contained in nested buses. An arrow next to a signal indicates that an input signal is a bus. To display the contents of that bus, click the arrow.

Signals in the bus are ordered from the top input port to the bottom input port. For a description of port order for various block orientations, see “Port Location After Rotating or Flipping”. To rearrange the signals in the output bus signal, use the **Up** and **Down** buttons. You can select multiple top-level contiguous signals in the **Signals in the bus** list to reorder or remove.

To find the source of any signal entering the block, select the signal in the **Signals in the bus** list and click the **Find** button. Simulink highlights the icon of the source, opening the subsystem that contains the signal source, if necessary.

**Filter by name — Search term used to filter input signals**
text

Search term used to filter a long list of input signals. Do not enclose the search term in quotation marks. The filter does a partial string search. To access filtering options, including using a regular expression to specify the search term, click the button to the right of the **Filter by name** edit box.

**Enable regular expression — Option to filter using regular expressions**
off (default) | on

Enable the use of MATLAB regular expressions for filtering signal names. For example, entering $t$ in the **Filter by name** edit box displays all signals whose names end with a lowercase $t$ (and their immediate parents). For details, see “Regular Expressions” (MATLAB).

**Dependencies**

To access this parameter, click the button to the right of the **Filter by name** edit box.
**Show filtered results as a flat list — Option to display filtered results as a flat list**

off (default) | on

Enable the flat list format that uses dot notation to reflect the hierarchy of bus signals based on the search text in the **Filter by name** edit box. By default, a tree list displays the filtered signals.

**Dependencies**

To access this parameter, click the button to the right of the **Filter by name** edit box.

**Output data type — Data type of output bus**

'Inherit: auto'(default)|'Bus: <object name>','<data type expression>

Data type of the output bus.

If you select 'Bus: <object name>', specify a bus object. The bus object must be accessible when you update the diagram or simulate the model. To define a bus object using the Bus Editor, click the **Show data type assistant** button, then click the **Edit** button.

If you select '<data type expression>', specify an expression that evaluates to a data type. Do not specify a bus object as the expression.

**Programmatic Use**

**Block Parameter:** OutDataTypeStr

**Type:** character vector

**Values:**

'Inherit: auto' | 'double' | 'single' | 'int8' | 'uint8' | 'int16' | 'uint16' | 'int32' | 'uint32' | 'boolean' | 'fixdt(1,16,0)' | 'fixdt(1,16,2^0,0)' | 'Enum: <class name>' | 'Bus: <object name>'

**Default:** 'Inherit: auto'

**Require input signal names to match signals above — Match input signal names to names listed in dialog box**

off (default) | on

Optional check that input signal names match the signal names in the Bus Creator Block Parameters dialog box.
Tip The Require input signal names to match signals above parameter might be removed in a future release. To enforce strong data typing, consider using the Override bus signal names from inputs parameter.

If you select Override bus signal names from inputs, Simulink software ignores the Require input signal names to match signals above setting.

If you change the Number of inputs programmatically, this parameter reverts to 'off'.

Rename selected signal — Name for currently selected signal

' ' (default) | character vector

Name the currently selected input signal. See “Signal Names and Labels” for signal name guidelines.

Dependencies

To display this parameter, enable the Require input signal names to match signals above parameter.

Override bus signal names from inputs — Bus input signal name override

on (default) | off

By default, the Bus Creator block overrides bus signal names from inputs.

To inherit bus signal names from a bus object, clear this parameter. Clearing the parameter:

- Enforces strong data typing.
- Avoids having to enter a signal name multiple times: in the bus object and in the model. Entering the name multiple times can accidentally create signal name mismatches.
- Supports the array of buses requirement to have consistent signal names across array elements.

Alternatively, you can enforce strong data typing by checking that input signal names match the bus object element names.

- Keep the Override bus signal names from inputs check box selected.
- Set the Element name mismatch parameter to error.
Dependencies

To display this parameter, set the **Output data type** parameter to a bus object.

**Programmatic Use**

**Block Parameter:** InheritFromInputs  
**Type:** character vector  
**Values:** 'on' | 'off'  
**Default:** 'on'

**Output as nonvirtual bus — Nonvirtual bus output**

off (default) | on

Nonvirtual bus output from the Bus Creator block.

All signals in a nonvirtual bus must have the same sample time, even if the elements of the associated bus object specify inherited sample times for some elements. Any operation resulting in a nonvirtual bus containing signals with different sample rates generates an error. To change the sample time of a signal or bus that has a different sample time than the other nonvirtual bus input signals, use a Rate Transition block. For details, see “Specify Bus Signal Sample Times”.

To generate code that uses a C structure to define the structure of the bus output by this block, enable this parameter.

**Dependencies**

To display this parameter, set the **Output data type** parameter to a bus object.

**Programmatic Use**

**Block Parameter:** NonVirtualBus  
**Type:** character vector  
**Values:** 'on' | 'off'  
**Default:** 'off'

**Block Characteristics**

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>bus</th>
<th>double</th>
<th>enumerated</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
<th>string</th>
</tr>
</thead>
</table>

### Extended Capabilities

#### C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

Actual data type or capability support depends on block implementation.

#### HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

To learn more about using buses for HDL code generation, see “Buses” (HDL Coder) and “Use Bus Signals to Improve Readability of Model and Generate HDL Code” (HDL Coder).

#### HDL Architecture
This block has a single, default HDL architecture.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>yes</td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>yes</td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
</tr>
</tbody>
</table>
### HDL Block Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConstrainedOutputPipeline</td>
<td>Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. For more details, see “ConstrainedOutputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td>InputPipeline</td>
<td>Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “InputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td>OutputPipeline</td>
<td>Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “OutputPipeline” (HDL Coder).</td>
</tr>
</tbody>
</table>

### PLC Code Generation
Generate Structured Text code using Simulink® PLC Coder™.

### Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

Actual data type or capability support depends on block implementation.

### See Also
Bus Assignment | Bus Selector | Bus to Vector | Out Bus Element

### Topics
- “Getting Started with Buses”
- “Simplify Bus Interfaces”
- “Nest Buses”
- “Bus-Capable Blocks”
- “Assign Signal Values to a Bus”
- “Composite Signals”
- “Specify Bus Signal Sample Times”
“Specify Bus Properties with Bus Objects”

*Introduced before R2006a*
Bus Selector

Select signals from incoming bus

Library:  
Simulink / Commonly Used Blocks  
Simulink / Signal Routing  
HDL Coder / Signal Routing

Description

The Bus Selector block outputs a specified subset of the elements of the bus at its input. The block can output the specified elements as separate signals or as a new bus.

By default, Simulink implicitly converts a nonbus signal to a bus to support connecting the signal to a Bus Selector block. To prevent Simulink from performing that conversion, set the “Non-bus signals treated as bus signals” diagnostic to warning or error.

When the block outputs multiple elements, it outputs each element from a separate port from top to bottom of the block. See “Port Location After Rotating or Flipping” for a description of the port order for various block orientations.

In the Simulink Editor, as you draw a new signal line close to output side of a Bus Selector block and all output ports are already connected, Simulink Editor:

- Adds a port
- Prompts you to specify the signal to be selected

You cannot use this automatic port addition approach in either of these cases if:

- There is no bus input signal connected to the Bus Selector block.
- You do not specify a signal in response to the prompt that appears when you draw a signal line close to the Bus Selector block icon.
- You select the **Output as virtual bus** parameter.

**Tip** For models that include buses composed of many elements, consider using the In Bus Element and Out Bus Element blocks. You can use these bus element port blocks instead
of Inport with Bus Selector blocks for inputs, and Outport with Bus Creator blocks for outputs.

Bus element port blocks:

- Reduce signal line complexity and clutter in a block diagram.
- Make it easier to change the interface incrementally.
- Allow access to a bus element closer to the point of usage, avoiding the use of a Bus Selector and Goto block configuration.

### Ports

#### Input

**Port_1 — Input bus**

real or complex values of any data type supported by Simulink except for arrays of buses

Input buses can have real or complex values of any data type supported by Simulink except for arrays of buses. To work with an array of buses, use a Selector block to select the index for the bus element that you want to use with the Bus Selector block. Then use that selected bus element with the Bus Selector block.

#### Output

**Port_1 — Selected bus elements of input bus**

standalone signal (default) | virtual bus | nonvirtual bus | array of buses

Selected bus elements of an input bus. For each output signal, this block uses a separate port from top to bottom of the block. See “Port Location After Rotating or Flipping” for a description of the port order for various block orientations.

If you select the **Output as virtual bus** parameter, the output bus is virtual. To produce nonvirtual bus output, insert a Signal Conversion block after the Bus Selector block. Set the Signal Conversion block **Output** parameter to **Nonvirtual bus** and use a Simulink.Bus bus object for the **Data type** parameter. For an example, see the Signal Conversion documentation.
Parameters

**Signals in the bus — Element signals in input bus**

- **List of signal names**

List of bus element signals of the input bus, from which to select signals to output. To select a signal to output, click the signal in the list and then click **Select**.

To refresh the display to reflect modifications to the bus connected to the block, click **Refresh**.

To find the source of a signal entering the block, select the signal in the list and click **Find**. The Simulink software highlights the signal source in the block diagram.

**Programmatic Use**

- **Block Parameter:** InputSignals
- **Type:** matrix
- **Values:** signal name
- **Default:** {'[]'}

**Filter by name — Filter set of displayed input signals by search term**

- **Text**

Specify a search term to use for filtering a long list of input signals. Do not enclose the search term in quotation marks. The filter does a partial string search. To access filtering options, including using a regular expression to specify the search term, click the button on the right side of the **Filter by name** edit box.

**Enable regular expression — Filter set of displayed input signals by regular expression**

- **Off (default) | on**

Enable the use of MATLAB regular expressions for filtering signal names. For example, entering `t$` in the **Filter by name** edit box displays all signals whose names end with a lowercase `t` (and their immediate parents). For details, see “Regular Expressions” (MATLAB).

**Dependencies**

To access this parameter, click the button on the right side of the **Filter by name** edit box.
Show filtered results as a flat list — Change appearance of filtered list
off (default) | on

By default, a tree displays the filtered signals, based on the search text in the Filter by name edit box. Select this parameter to use a flat list format that uses dot notation to reflect the hierarchy of buses.

Dependencies

To access this parameter, click the button on the right side of the Filter by name edit box

Selected signals — Selected bus elements
list of signal names

If an output signal listed in the Selected signals list box is not an input to the Bus Selector block, the signal name starts with three question marks (???).

You can change the list by using the Up, Down, and Remove buttons. To save your changes, click Apply. You can select multiple contiguous signals to move or remove. You cannot rearrange leaf signals within a bus. For example, you can move bus Bus1 up or down in the list, but you cannot reorder any of the bus elements of Bus1. Port connectivity is maintained when you change the signal order.

Programmatic Use
Block Parameter: OutputSignals
Type: character vector
Values: character vector in the form of 'signal1,signal2'
Default: none

Output as virtual bus — Output selected elements as bus
on (default) | off

By default, the block outputs the selected elements as standalone signals, each from an output port that is labeled with the corresponding bus element name. To output the selected bus element signals as a bus, select this parameter.

The output bus is virtual. To produce nonvirtual bus output, insert a Signal Conversion block after the Bus Selector block. Set the Signal Conversion block Output parameter to Nonvirtual bus and use a Simulink.Bus bus object for the Data type parameter. For an example, see the Signal Conversion documentation.
When the Selected signals list includes only one signal and you enable Output as virtual bus, then if the selected signal is:

- A nonbus signal, it is treated as a nonbus signal (it is not wrapped in a bus).
- A bus, then the output is that bus.

Programmatic Use

**Block Parameter:** OutputAsBus

**Type:** character vector

**Values:** 'on' | 'off'

**Default:** 'off'

## Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>bus</th>
<th>double</th>
<th>enumerated</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
<th>string</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Actual data type or capability support depends on block implementation.
HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

To learn more about using buses for HDL code generation, see “Buses” (HDL Coder) and “Use Bus Signals to Improve Readability of Model and Generate HDL Code” (HDL Coder).

HDL Architecture
This block has a single, default HDL architecture.

HDL Block Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConstrainedOutputPipeline</td>
<td>Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. For more details, see “ConstrainedOutputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td>InputPipeline</td>
<td>Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “InputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td>OutputPipeline</td>
<td>Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “OutputPipeline” (HDL Coder).</td>
</tr>
</tbody>
</table>

PLC Code Generation
Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

Actual data type or capability support depends on block implementation.
See Also
Bus Assignment | Bus Creator | Bus to Vector | Out Bus Element

Topics
“Getting Started with Buses”
“Simplify Bus Interfaces”
“Nest Buses”
“Bus-Capable Blocks”
“Composite Signals”

Introduced before R2006a
Bus to Vector

Convert virtual bus to vector

Library: Simulink / Signal Attributes
HDL Coder / Signal Attributes

Description

The Bus to Vector block converts a virtual bus to a vector signal. The input bus must consist of scalars or 1-D, row, or column vectors that have the same data type, signal type, and sampling mode. If the input bus contains row or column vectors, the output is a row or column vector, respectively. Otherwise, the output is a 1-D array.

Use the Bus to Vector block only to replace an implicit bus-to-vector conversion with an explicit conversion. To identify and correct buses used as vectors without manually inserting Bus to Vector blocks, you can use the Simulink Model Advisor “Check bus signals treated as vectors” on page 10-29 check. Alternatively, you can use the Simulink.BlockDiagram.addBusToVector function, which automatically inserts Bus to Vector blocks wherever needed.

Note If you use Save As to save a model in a version of the Simulink product before R2007a, a null subsystem that outputs nothing replaces each Bus to Vector block. Before you can use the model, reconnect or otherwise correct each path that used to contain a Bus to Vector block but now is interrupted by a null subsystem.

Ports

Input

Port_1 — Accept signal to convert to vector
virtual bus | nonbus signal
The input bus signals must consist of scalars or 1-D, row, or column vectors that have the same data type, signal type, and sampling mode. If the input is a nonbus signal, the block does no conversion.

**Output**

*Port_1 — Output vector signal*

*vector | 1-D array*

Output a vector signal, based on input bus signal. If the input bus contains row or column vectors, the block output is a row or column vector, respectively. Otherwise, the output is a 1-D array.

**Block Characteristics**

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>bus</th>
<th>double</th>
<th>enumerated</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Extended Capabilities**

**C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

Actual data type or capability support depends on block implementation.
**HDL Code Generation**
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

To learn more about using buses for HDL code generation, see “Buses” (HDL Coder) and “Use Bus Signals to Improve Readability of Model and Generate HDL Code” (HDL Coder).

**HDL Architecture**
This block has a single, default HDL architecture.

**HDL Block Properties**

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConstrainedOutput Pipeline</td>
<td>Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. For more details, see “ConstrainedOutputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td>InputPipeline</td>
<td>Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “InputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td>OutputPipeline</td>
<td>Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “OutputPipeline” (HDL Coder).</td>
</tr>
</tbody>
</table>

**Fixed-Point Conversion**
Design and simulate fixed-point systems using Fixed-Point Designer™.

Actual data type or capability support depends on block implementation.

**See Also**
Bus Creator | Bus Selector | Data Type Conversion | Simulink.BlockDiagram.addBusToVector
Topics
“Correct Buses Used as Vectors”
“Types of Composite Signals”

Introduced in R2007a
### Callback Button

**Execute MATLAB code based on user input**

**Library:** Simulink / Dashboard

### Description

The Callback Button block executes MATLAB code based on user input. The Callback Button block reacts to clicks and presses from the user. You can specify separate code to execute for each action. The Callback Button block repeats the code specified for the press action at a specified rate as long as you continue to press the button.

The Callback Button block registers a click when you release the left mouse button with the cursor on the Callback Button. The code for a press executes when you click the Callback Button and hold for the specified **Press Delay**.

Double-clicking the Callback Button block does not open its dialog box during simulation and when the block is selected. To edit the block's parameters, you can use the **Property Inspector**, or you can right-click the block and select **Block Parameters** from the context menu.

### Parameters

**Button Text — Text on button**

'Callback Button' (default) | character array

Text that appears on the button.

**Mouse Action — Mouse action to run code**

'ClickFcn' (default) | 'PressFcn'

Mouse action that causes the MATLAB code to execute.

- Select **ClickFcn** to view and edit the code that executes when the Callback Button block is clicked.
• Select PressFcn to view and edit the code that executes when the button is pressed.

**MATLAB Code — Code to execute based on user input**

empty (default) | MATLAB Code

MATLAB code that executes based on user input.

**Dependency**

The MATLAB code displayed depends on the Mouse Action parameter selection.

• Select ClickFcn to view and edit the code that executes when the Callback Button block is clicked.

• Select PressFcn to view and edit the code that executes when the button is pressed.

**Press Delay (ms) — Time to hold button for press**

500 (default) | scalar

Amount of time required to cause the PressFcn code to execute.

**Dependency**

**Press Delay (ms)** is only visible when PressFcn is selected as the Mouse Action.

**Repeat Interval (ms) — Time interval to repeat PressFcn code**

0 (default) | scalar

Time interval after which the PressFcn code executes again if the Callback Button block is still pressed.

**Dependency**

**Repeat Interval (ms)** is only visible when PressFcn is selected as the Mouse Action.

**Block Characteristics**

<table>
<thead>
<tr>
<th>Data Types</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
</tr>
</tbody>
</table>

1-111
<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multidimensional Signals</td>
<td>no</td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
</tr>
</tbody>
</table>

**Extended Capabilities**

**C/C++ Code Generation**
Generate C and C++ code using Simulink® Coder™.

Dashboard blocks are ignored for code generation.

**See Also**
Push Button

**Topics**
“Tune and Visualize Your Model with Dashboard Blocks”
“Decide How to Visualize Simulation Data”

**Introduced in R2017b**
C Caller

Integrate C code in Simulink
Library: Simulink / User-Defined Functions

Description

The C Caller block integrates your external C code into Simulink. This block imports and lists the functions in your external C code, and allows you to select your resolved C functions to integrate in your Simulink models. The C Caller block standalone supports code generation. For more complex models, code generation depends on the capabilities of your Simulink model.

To use the C Caller block, define your source code and any supporting files using Simulation Target under Configuration Parameters. Then, bring a C Caller block to the Simulink canvas, using Library Browser > Simulink > User Defined Functions. To change the defined source code file and its dependencies, go to Simulation Target tab in Configuration Parameters by clicking the from the block dialog. After changing your source code or any of its dependencies, refresh the list of functions by clicking the on the block dialog. To browse and modify the list of functions in your source code, use the icon to access your header files.

Ports

Input

Input argument — Input argument for the corresponding C caller block
scalar | vector | matrix
Input argument to the C Caller block.

Number of input arguments and their names are inferred through the selected function in your external C code. To receive data to a C Caller block, connect an input signal to the input ports.

Input label has the same name as your input argument unless changed by editing the Label column under Port Specification from the Block Dialog. If you rename the label to an input port, the C Caller block changes the name of the port.

For input variables, you can change the input scope to parameters or constants using the Scope column.

Data types supported in MATLAB, but not supported in Simulink cannot be passed between C Caller and other MATLAB blocks.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | Boolean | fixed point | enumerated | bus

Output

Output argument — Output argument for the corresponding C Caller block

scalar | vector | matrix

Output argument from the C Caller block.

Number of output arguments and their names are inferred through the selected function in your external C code. To send data from your C Caller block, connect a block to the output port of your C Caller block.

Output port label has the same name as your output argument unless you change it by editing the Label column under Port Specification from the Block Dialog. If you rename the label to an input port, the C Caller block changes the name of the port.

Data types supported in MATLAB, but not supported in Simulink cannot be passed between C Caller and other MATLAB blocks.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | Boolean | enumerated | bus
Parameters

**Function name** — Name of the functions in source code
character vector

The C Caller block in your model imports all functions in your external source code, and shows the names of functions under Function name in the block dialog. To select and use a function in your block, confirm that the function names appear at the Function name table. In case you are missing one of the functions, reload the source code by clicking on the block dialog. To change the names of your functions, modify your source code and click the refresh button to reload.

**Port specification** — Specify port properties
string

The Port specification table indicates the attributes of each input and output element for the selected block. These properties include Argument name, scope, input/output label, type and size of the input/output variables. Argument name, scope, type, and size are inferred from your source code. If the scope is an input, you can modify this variable to a parameter or a constant.

**Arg name** — Demonstrates the variable name inferred from your source code.

**Scope** — Indicates the role of the variables from your source code. If the variable is an input argument in the C Caller block source code, you can change the scope type to a constant or a parameter. If the variable is an output argument in the source code, you cannot change the scope type.

**Label** — Labels the input or output variable for the Simulink model. You can change the labels using this table. If the scope is a parameter, enter the parameter name in this field. If the scope is a constant, enter the constant value.

**Type** — Indicates the data type coming from the ports.

**Size** — Indicates the size of the input and output data.
Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean(^a)</th>
<th>bus(^a)</th>
<th>double(^a)</th>
<th>enumerated(^a)</th>
<th>fixed point(^a)</th>
<th>integer(^a)</th>
<th>single(^a)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>limited(^a)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\(^a\) Actual data type or capability support depends on block implementation.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

Actual data type or capability support depends on block implementation.

Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

See Also
MATLAB Function | MATLAB System | S-Function | S-Function Builder

Topics
“Integrate C Code Using C Caller Blocks”
Introduced in R2018b
Check Box

Select parameter or variable value

Library: Simulink / Dashboard

Description

The Check Box block allows you to set the value of a parameter or variable during simulation by checking or clearing the box. Use the Check Box block with other Dashboard blocks to create an interactive dashboard for your model.

Double-clicking the Check Box block does not open its dialog box during simulation and when the block is selected. To edit the block's parameters, you can use the Property Inspector, or you can right-click the block and select Block Parameters from the context menu.

Connecting Dashboard Blocks

Dashboard blocks do not use ports to make connections. To connect Dashboard blocks to variables and block parameters in your model, use connect mode. Connect mode facilitates the process of connecting Dashboard blocks in your model, especially when you want to connect multiple blocks at once. If you only want to connect a single Dashboard block, you can also use the Connection table in the block dialog box to make the connection.

Tip To facilitate understanding and debugging your model, you can modify Dashboard block connections in your model during normal and accelerator mode simulations.

Note Dashboard blocks cannot connect to variables until you update your model diagram. To connect Dashboard blocks to variables or modify variable values between opening your model and running a simulation, update your model diagram using Ctrl+D.
To enter connect mode, click the **Connect** button that appears above your unconnected Dashboard block when you pause on it.

In connect mode, when you select one or more signals or blocks, a list of parameters and signals available for connection appears. Select a signal or parameter from the list to connect the selected Dashboard block. To connect another Dashboard block, pause on the block and click the **Connect** button above it. Then, make a selection of signals and blocks in your model, and choose a signal or parameter to connect.
When you finish connecting the Dashboard blocks in your model, click the **Exit** button in the upper-right of the canvas to exit connect mode.

**Tip** You can hide the message shown on unconnected blocks using the `set_param` function with the `ShowInitialText` block parameter.

### Parameter Logging

Tunable parameters connected to Dashboard blocks are logged to the Simulation Data Inspector, where you can view parameter values along with logged signal data. You can access logged parameter data in the MATLAB workspace by exporting the parameter data from the Simulation Data Inspector UI or by using the `Simulink.sdi.exportRun` function. For more information about exporting data with the Simulation Data Inspector UI, see “Export Data from the Simulation Data Inspector”. The parameter data is stored in a `Simulink.SimulationData.Parameter` object, accessible as an element in the exported `Simulink.SimulationData.Dataset`. 
Limitations

- Dashboard blocks can only connect to real scalar signals.
- You cannot use the Connection table to connect a Dashboard block to a block that is commented out. When you connect a Dashboard block to a commented block using connect mode, the Dashboard block does not display the connected value until you uncomment the block.
- Dashboard blocks cannot connect to signals inside referenced models.
- When you simulate a model hierarchy, Dashboard blocks inside referenced models do not update.
- Dashboard blocks do not support rapid accelerator simulation.
- Some signals are not available during accelerator mode simulations. To view such signals on a Dashboard block, mark the signal for logging.
- Parameters specified by indexing a variable array do not appear in the Connection table. For example, a block parameter defined as engine(1) using the variable engine does not appear in the table.

To access the parameter in the Connection table, assign the indexed value to a scalar variable, such as engine_1. Then, use the scalar variable to define the block parameter.

- When you connect a Dashboard block to a variable or parameter during simulation, the data for that variable or parameter does not get logged to the Simulation Data Inspector. To log variable and parameter data to the Simulation Data Inspector, connect the Dashboard block to the variable or parameter prior to simulation.
- When you simulate a model in external mode with the Default parameter behavior set to Inlined, Dashboard blocks can appear to change parameter and variable values. However, the change does not propagate to the simulation. For example, the Gain block displays changes made to its Gain parameter using the Dashboard blocks, but the Gain value used in the simulation does not change.

Parameters

Connection — Select a variable or block parameter to connect
variable and parameter connection options

Select the variable or block parameter to control using the Connection table. Populate the Connection table by selecting one or more blocks in your model. Select the radio
button next to the variable or parameter you want to control, and click **Apply**. To
facilitate understanding and debugging your model, you can connect Dashboard blocks to
variables and parameters in your model during simulation.

**Note** To see workspace variables in the connection table, update the model diagram
using **Ctrl+D**.

**Programmatic Use**

To programmatically connect a Dashboard block to a tunable parameter or a variable, use
a `Simulink.HMI.ParamSourceInfo` object. The `Simulink.HMI.ParamSourceInfo`
object contains four properties. Some of the properties apply to connecting Dashboard
blocks to parameters, and some apply to connecting Dashboard blocks to variables. Not
all fields have a value for a connection because a given Dashboard block connects to
either a parameter or a variable.

**Block Parameter:** Binding

**Type:** `Simulink.HMI.ParamSourceInfo`

**Default:** `[]`

**Check Box Label — Description of check box action**

**Label (default) | character vector**

Label that appears next to the check box. You can use the **Check Box Label** to describe
what happens in your model when you check the check box.

**Example:** Enable sound

**Programmatic Use**

**Block Parameter:** Label

**Type:** `character vector`

**Default:** `'Label'`

**Checked/Unchecked Values — Values to assign for checked and unchecked
states**

**Unchecked: 0 Checked: 1 (default) | scalar**

Value to assign to the connected variable or parameter for the checked and unchecked
block states.

- **Unchecked** — Value assigned to the connected parameter when the Check Box block
  is not checked.
• Checked — Value assigned to the connected parameter when the Check Box block is checked.

**Programmatic Use**

You set the Unchecked and Checked values programmatically using a 2-by-1 vector that contains the Unchecked and Checked values, in that order.

**Block Parameter: Values**

Type: 2x1 vector

Default: [0 1]

**Label — Block label position**

Hide (default) | Bottom | Top

Position of the block label. When the block is connected to an element in the model, the label is the name of the connected element.

**Programmatic Use**

**Block Parameter: LabelPosition**

Type: character vector

Values: 'Hide' | 'Bottom' | 'Top'

Default: 'Hide'

**Block Characteristics**

<table>
<thead>
<tr>
<th>Data Types</th>
<th>double</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>no</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

Dashboard blocks are ignored for code generation.

See Also
Rocker Switch | Rotary Switch | Slider Switch | Toggle Switch

Topics
“Tune and Visualize Your Model with Dashboard Blocks”
“Decide How to Visualize Simulation Data”

Introduced in R2017b
Check Discrete Gradient

Check that absolute value of difference between successive samples of discrete signal is less than upper bound

**Library:** Simulink / Model Verification
HDL Coder / Model Verification

**Description**

The Check Discrete Gradient block checks each signal element at its input to determine whether the absolute value of the difference between successive samples of the element is less than an upper bound. Specify the value of the upper bound (1 by default) by setting the **Maximum gradient** parameter. If the verification condition is true, the block does nothing. Otherwise, the block halts the simulation, by default, and displays an error in the Diagnostic Viewer.

Enable or disable all model verification blocks by changing the **Model Verification** block enabling setting in the Configuration Parameters.

Use the blocks in the Model Verification library to facilitate creation of self-validating models. For example, you can use model verification blocks to test that signals do not exceed specified limits during simulation. When you are satisfied that a model is correct, you can turn error checking off by disabling the verification blocks. You do not have to remove them from the model. If you need to modify a model, you can temporarily turn the verification blocks back on to ensure that your changes do not break the model.

**Ports**

**Input**

*Port_1 — Input signal*

scalar | vector | matrix
Input signal the block checks to determine if the difference of each element between successive samples is less than the upper bound. Specify the upper bound by setting the **Maximum gradient** parameter.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

**Output**

**Port_1 — Assertion output signal**

scalar

Output signal at each time step that is true (1) if the assertion succeeds, and false (0) if the assertion fails. If, in the Configuration Parameters, you select **Implement logic signals as Boolean data**, then the output data type is a Boolean. Otherwise the data type of the signal is a double.

**Dependencies**

To enable this output port, select the **Output assertion signal** parameter check box.

Data Types: double | Boolean

**Parameters**

**Maximum gradient — Upper bound of allowed differences**

1 (default) | scalar

Specify the upper bound on the allowed gradient of the input signal.

**Command-Line Information**

Parameter: gradient

Type: character vector

Values: real scalar

Default: '1'

**Enable assertion — Enable or disable the check**

on (default) | off

Clearing this check box disables the block and causes the model to behave as if the block does not exist. You can set the **Model Verification block enabling** setting in the
Configuration Parameters to enable or disable all model verification blocks in a model regardless of the setting of this option.

**Command-Line Information**
- **Parameter:** enabled
- **Type:** character vector
- **Values:** 'on' | 'off'
- **Default:** 'on'

**Simulation callback when assertion fails (optional) — Expression to evaluate when assertion fails**
- ' ' (default) | MATLAB expression

Specify a MATLAB expression to evaluate when the assertion fails. Because the expression is evaluated in the MATLAB workspace, define all variables used in the expression in that workspace.

**Command-Line Information**
- **Parameter:** callback
- **Type:** character vector
- **Values:** MATLAB expression
- **Default:** ' '

**Stop simulation when assertion fails — Halt simulation when check fails**
- on (default) | off

Select this check box to indicate that the block halts simulation when the check fails. Clear to indicate that the software displays a warning and continues the simulation.

**Command-Line Information**
- **Parameter:** stopWhenAssertionFail
- **Type:** character vector
- **Values:** 'on' | 'off'
- **Default:** 'on'

**Output assertion signal — Create output signal**
- off (default) | on

Selecting this check box causes the block to output a Boolean signal that is true (1) at each time step if the assertion succeeds and false (0) if the assertion fails. The data type of the output signal is Boolean if you have selected the **Implement logic signals as Boolean data** check box on the Configuration Parameters dialog box. Otherwise the data type of the output signal is double.
Command-Line Information
Parameter: export
Type: character vector
Values: 'on' | 'off'
Default: 'off'

Select icon type — Select icon type
graphic (default) | text

Specify the type of icon used to display this block in a block diagram. The graphic option displays a graphical representation of the assertion condition on the icon. The text option displays a mathematical expression that represents the assertion condition. If the icon is too small to display the expression, the text icon displays an exclamation point. To see the expression, enlarge the block.

Command-Line Information
Parameter: icon
Type: character vector
Values: 'graphic' | 'text'
Default: 'graphic'

Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>double</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

For information about how Simulink Coder generated code handles Model Verification blocks, see “Debug” (Simulink Coder).

Not recommended for production code.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture

This block can be used to check whether the difference between successive samples is less than the upper bound during simulation, but is not included in the generated code.

PLC Code Generation
Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

See Also
Check Dynamic Gap | Check Dynamic Range

Introduced before R2006a
Check Dynamic Gap

Check that gap of possibly varying width occurs in range of signal's amplitudes

Library: Simulink / Model Verification
HDL Coder / Model Verification

Description

The Check Dynamic Gap block checks that a gap of possibly varying width occurs in the range of a signal's amplitudes. The test signal is connected to \( u \). The inputs \( \text{min} \) and \( \text{max} \) specify the lower and upper bounds of the dynamic gap. If the verification condition is true, the block does nothing. If not, the block halts the simulation, by default, and displays an error message.

Use the blocks in the Model Verification library to facilitate creation of self-validating models. For example, you can use model verification blocks to test that signals do not exceed specified limits during simulation. When you are satisfied that a model is correct, you can turn error checking off by disabling the verification blocks. You do not have to remove them from the model. If you need to modify a model, you can temporarily turn the verification blocks back on to ensure that your changes do not break the model.

Ports

Input

\( \text{max} \) — Upper bound of dynamic gap

scalar | vector | matrix

Signal specifying the upper bound of the gap. All three input signals must be the same data type.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated
**min — Lower bound of dynamic gap**
scalar | vector | matrix

Signal specifying the lower bound of the gap. All three input signals must be the same data type and dimension.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated

**u — Input signal**
scalar | vector | matrix

Input signal checked for a gap of width specified by max and min. All three input signals must be the same data type and dimension.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated

**Output**

**Port_1 — Assertion output signal**
scalar

Output signal at each time step that is true (1) if the assertion succeeds, and false (0) if the assertion fails. If, in the Configuration Parameters, you select **Implement logic signals as Boolean data**, then the output data type is a Boolean. Otherwise the data type of the signal is a double.

**Dependencies**

To enable this output port, select the **Output assertion signal** parameter check box.

Data Types: double | Boolean

**Parameters**

**Enable assertion — Enable or disable the check**
on (default) | off

Clearing this check box disables the block and causes the model to behave as if the block does not exist. You can set the **Model Verification block enabling** setting in the
Configuration Parameters to enable or disable all model verification blocks in a model regardless of the setting of this option.

**Command-Line Information**
**Parameter:** enabled
**Type:** character vector
**Values:** 'on' | 'off'
**Default:** 'on'

**Simulation callback when assertion fails (optional) — Expression to evaluate when assertion fails**
'
'(default) | MATLAB expression

Specify a MATLAB expression to evaluate when the assertion fails. Because the expression is evaluated in the MATLAB workspace, define all variables used in the expression in that workspace.

**Command-Line Information**
**Parameter:** callback
**Type:** character vector
**Values:** MATLAB expression
**Default:** '
'

**Stop simulation when assertion fails — Halt simulation when check fails**
on (default) | off

Select this check box to indicate that the block halts simulation when the check fails. Clear to indicate that the software displays a warning and continues the simulation.

**Command-Line Information**
**Parameter:** stopWhenAssertionFail
**Type:** character vector
**Values:** 'on' | 'off'
**Default:** 'on'

**Output assertion signal — Create output signal**
off (default) | on

Selecting this check box causes the block to output a Boolean signal that is true (1) at each time step if the assertion succeeds and false (0) if the assertion fails. The data type of the output signal is **Boolean** if you have selected the **Implement logic signals as Boolean data** check box on the Configuration Parameters dialog box. Otherwise the data type of the output signal is **double**.
Command-Line Information
Parameter: export
Type: character vector
Values: 'on' | 'off'
Default: 'off'

Select icon type — Select icon type
graphic (default) | text

Specify the type of icon used to display this block in a block diagram. The graphic option displays a graphical representation of the assertion condition on the icon. The text option displays a mathematical expression that represents the assertion condition. If the icon is too small to display the expression, the text icon displays an exclamation point. To see the expression, enlarge the block.

Command-Line Information
Parameter: icon
Type: character vector
Values: 'graphic' | 'text'
Default: 'graphic'

Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>double</th>
<th>enumerated</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

For information about how Simulink Coder generated code handles Model Verification blocks, see “Debug” (Simulink Coder).

Not recommended for production code.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture
This block can be used to check the gap in the varying signal amplitudes, but it is not included in the generated code.

PLC Code Generation
Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

See Also
Check Dynamic Lower Bound | Check Dynamic Range | Check Dynamic Upper Bound

Introduced before R2006a
Check Dynamic Lower Bound

Check that one signal is always less than another signal

Library:  Simulink / Model Verification
          HDL Coder / Model Verification

Description

The Check Dynamic Lower Bound block checks that the amplitude of a reference signal, \( \text{min} \), is less than the amplitude of a test signal, \( \text{u} \), at the current time step. If the verification condition is true, the block does nothing. If not, the block halts the simulation, by default, and displays an error message.

Use the blocks in the Model Verification library to facilitate creation of self-validating models. For example, you can use model verification blocks to test that signals do not exceed specified limits during simulation. When you are satisfied that a model is correct, you can turn error checking off by disabling the verification blocks. You do not have to remove them from the model. If you need to modify a model, you can temporarily turn the verification blocks back on to ensure that your changes do not break the model.

Ports

Input

\( \text{min} \) — Lower bound of check

*scalar | vector | matrix*

Signal specifying the lower bound of the check against the input signal \( \text{u} \) amplitude. Signal data type and dimension must be the same as \( \text{u} \).

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated

\( \text{u} \) — Input signal

*scalar | vector | matrix*
Input signal checked against the lower bound specified by \textbf{min}. Both input signals must be the same data type and dimension.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated

\textbf{Output}

\textbf{Port_1 — Assertion output signal}
scalar

Output signal at each time step that is true (1) if the assertion succeeds, and false (0) if the assertion fails. If, in the Configuration Parameters, you select \textbf{Implement logic signals as Boolean data}, then the output data type is a Boolean. Otherwise the data type of the signal is a double.

\textbf{Dependencies}

To enable this output port, select the \textbf{Output assertion signal} parameter check box.

Data Types: double | Boolean

\textbf{Parameters}

\textbf{Enable assertion — Enable or disable the check}
on (default) | off

Clearing this check box disables the block and causes the model to behave as if the block does not exist. You can set the \textbf{Model Verification block enabling} setting in the Configuration Parameters to enable or disable all model verification blocks in a model regardless of the setting of this option.

\textbf{Command-Line Information}

Parameter: enabled
Type: character vector
Values: 'on' | 'off'
Default: 'on'

\textbf{Simulation callback when assertion fails (optional) — Expression to evaluate when assertion fails}
' ' (default) | MATLAB expression
Specify a MATLAB expression to evaluate when the assertion fails. Because the expression is evaluated in the MATLAB workspace, define all variables used in the expression in that workspace.

**Command-Line Information**
**Parameter:** callback  
**Type:** character vector  
**Values:** MATLAB expression  
**Default:** ''

**Stop simulation when assertion fails — Halt simulation when check fails**
**on** (default) | **off**

Select this check box to indicate that the block halts simulation when the check fails. Clear to indicate that the software displays a warning and continues the simulation.

**Command-Line Information**
**Parameter:** stopWhenAssertionFail  
**Type:** character vector  
**Values:** 'on' | 'off'  
**Default:** 'on'

**Output assertion signal — Create output signal**
**off** (default) | **on**

Selecting this check box causes the block to output a Boolean signal that is true (1) at each time step if the assertion succeeds and false (0) if the assertion fails. The data type of the output signal is Boolean if you have selected the **Implement logic signals as Boolean data** check box on the Configuration Parameters dialog box. Otherwise the data type of the output signal is double.

**Command-Line Information**
**Parameter:** export  
**Type:** character vector  
**Values:** 'on' | 'off'  
**Default:** 'off'

**Select icon type — Select icon type**
**graphic** (default) | **text**

Specify the type of icon used to display this block in a block diagram. The **graphic** option displays a graphical representation of the assertion condition on the icon. The **text** option displays a mathematical expression that represents the assertion condition. If the
icon is too small to display the expression, the text icon displays an exclamation point. To see the expression, enlarge the block.

**Command-Line Information**
**Parameter:** icon  
**Type:** character vector  
**Values:** 'graphic' | 'text'  
**Default:** 'graphic'

**Block Characteristics**

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>double</th>
<th>enumerated</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Extended Capabilities**

**C/C++ Code Generation**
Generate C and C++ code using Simulink® Coder™.

For information about how Simulink Coder generated code handles Model Verification blocks, see “Debug” (Simulink Coder).

**HDL Code Generation**
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.
HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

**HDL Architecture**

This block can be used to check the lower bound of a test signal as compared to the input, but it is not included in the generated code.

**PLC Code Generation**

Generate Structured Text code using Simulink® PLC Coder™.

**Fixed-Point Conversion**

Design and simulate fixed-point systems using Fixed-Point Designer™.

**See Also**

*Introduced before R2006a*
Check Dynamic Range

Check that signal falls inside range of amplitudes that varies from time step to time step

**Library:**
- Simulink / Model Verification
- HDL Coder / Model Verification

**Description**

The Check Dynamic Range block checks that a test signal falls inside a range of amplitudes at each time step. The width of the range can vary from time step to time step. The input labeled $u$ is the test signal. The inputs labeled $\text{min}$ and $\text{max}$ are the lower and upper bounds of the valid range at the current time step. If the verification condition is true, the block does nothing. If not, the block halts the simulation, by default, and displays an error message.

Use the blocks in the Model Verification library to facilitate creation of self-validating models. For example, you can use model verification blocks to test that signals do not exceed specified limits during simulation. When you are satisfied that a model is correct, you can turn error checking off by disabling the verification blocks. You do not have to remove them from the model. If you need to modify a model, you can temporarily turn the verification blocks back on to ensure that your changes do not break the model.

**Ports**

**Input**

$max$ — **Upper bound of dynamic range check**

* scalar | vector | matrix

Signal specifying the upper bound of the range that the block checks the input signal $u$ amplitude. All three input signals must be the same data type.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated
**min — Lower bound of dynamic range check**

scalar | vector | matrix

Signal specifying the lower bound of the range that the block checks the input signal \( u \) amplitude. All three input signals must be the same data type and dimension.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated

**u — Input signal**

scalar | vector | matrix

Input signal checked against the range specified by \( \text{max} \) and \( \text{min} \). All three input signals must be the same data type and dimension.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated

**Output**

**Port_1 — Assertion output signal**

scalar

Output signal at each time step that is true (1) if the assertion succeeds, and false (0) if the assertion fails. If, in the Configuration Parameters, you select **Implement logic signals as Boolean data**, then the output data type is a **Boolean**. Otherwise the data type of the signal is a **double**.

**Dependencies**

To enable this output port, select the **Output assertion signal** parameter check box.

Data Types: double | Boolean

**Parameters**

**Enable assertion — Enable or disable the check**

on (default) | off

Clearing this check box disables the block and causes the model to behave as if the block does not exist. You can set the **Model Verification block enabling** setting in the
Configuration Parameters to enable or disable all model verification blocks in a model regardless of the setting of this option.

**Command-Line Information**

**Parameter:** enabled
**Type:** character vector
**Values:** 'on' | 'off'
**Default:** 'on'

**Simulation callback when assertion fails (optional) — Expression to evaluate when assertion fails**

' ' (default) | MATLAB expression

Specify a MATLAB expression to evaluate when the assertion fails. Because the expression is evaluated in the MATLAB workspace, define all variables used in the expression in that workspace.

**Command-Line Information**

**Parameter:** callback
**Type:** character vector
**Values:** MATLAB expression
**Default:** '

**Stop simulation when assertion fails — Halt simulation when check fails**

on (default) | off

Select this check box to indicate that the block halts simulation when the check fails. Clear to indicate that the software displays a warning and continues the simulation.

**Command-Line Information**

**Parameter:** stopWhenAssertionFail
**Type:** character vector
**Values:** 'on' | 'off'
**Default:** 'on'

**Output assertion signal — Create output signal**

off (default) | on

Selecting this check box causes the block to output a Boolean signal that is true (1) at each time step if the assertion succeeds and false (0) if the assertion fails. The data type of the output signal is Boolean if you have selected the **Implement logic signals as Boolean data** check box on the Configuration Parameters dialog box. Otherwise the data type of the output signal is double.
**Command-Line Information**
 Parameter: export  
Type: character vector  
Values: 'on' | 'off'  
Default: 'off'

**Select icon type** — Select icon type

- graphic (default) | text

Specify the type of icon used to display this block in a block diagram. The **graphic** option displays a graphical representation of the assertion condition on the icon. The **text** option displays a mathematical expression that represents the assertion condition. If the icon is too small to display the expression, the text icon displays an exclamation point. To see the expression, enlarge the block.

**Command-Line Information**
 Parameter: icon  
Type: character vector  
Values: 'graphic' | 'text'  
Default: 'graphic'

### Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>double</th>
<th>enumerated</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

For information about how Simulink Coder generated code handles Model Verification blocks, see “Debug” (Simulink Coder).

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

**HDL Architecture**

This block can be used to check whether a test signal falls within a range of amplitudes, but it is not included in the generated code.

PLC Code Generation
Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

See Also

Introduced before R2006a
Check Dynamic Upper Bound

Check that one signal is always greater than another signal

Library: Simulink / Model Verification
        HDL Coder / Model Verification

Description

The Check Dynamic Upper Bound block checks that the amplitude of a reference signal, \( \text{max} \), is greater than the amplitude of a test signal, \( u \), at the current time step. If the verification condition is true, the block does nothing. If not, the block halts the simulation, by default, and displays an error message.

Use the blocks in the Model Verification library to facilitate creation of self-validating models. For example, you can use model verification blocks to test that signals do not exceed specified limits during simulation. When you are satisfied that a model is correct, you can turn error checking off by disabling the verification blocks. You do not have to remove them from the model. If you need to modify a model, you can temporarily turn the verification blocks back on to ensure that your changes do not break the model.

Note For information about how Simulink Coder generated code handles Model Verification blocks, see “Debug” (Simulink Coder).

Ports

Input

\( \text{max} \) — Upper bound of range check
scalar | vector | matrix

Signal specifying the lower bound of the range that the block checks the input signal \( u \) amplitude. Signal data type and dimension must be the same as \( u \).
Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated

**u — Input signal**
scalar | vector | matrix

Input signal checked against the lower bound specified by `min`. Both input signals must be the same data type and dimension.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated

**Output**

**Port_1 — Assertion output signal**
scalar

Output signal at each time step that is true (1) if the assertion succeeds, and false (0) if the assertion fails. If, in the Configuration Parameters, you select **Implement logic signals as Boolean data**, then the output data type is a Boolean. Otherwise the data type of the signal is a double.

**Dependencies**

To enable this output port, select the **Output assertion signal** parameter check box.

Data Types: double | Boolean

**Parameters**

**Enable assertion — Enable or disable the check**
on (default) | off

Clearing this check box disables the block and causes the model to behave as if the block does not exist. You can set the **Model Verification block enabling** setting in the Configuration Parameters to enable or disable all model verification blocks in a model regardless of the setting of this option.

**Command-Line Information**

**Parameter:** enabled

**Type:** character vector
Values: 'on' | 'off'
Default: 'on'

Simulation callback when assertion fails (optional) — Expression to evaluate when assertion fails
' ' (default) | MATLAB expression

Specify a MATLAB expression to evaluate when the assertion fails. Because the expression is evaluated in the MATLAB workspace, define all variables used in the expression in that workspace.

Command-Line Information
Parameter: callback
Type: character vector
Values: MATLAB expression
Default: ' '

Stop simulation when assertion fails — Halt simulation when check fails
on (default) | off

Select this check box to indicate that the block halts simulation when the check fails. Clear to indicate that the software displays a warning and continues the simulation.

Command-Line Information
Parameter: stopWhenAssertionFail
Type: character vector
Values: 'on' | 'off'
Default: 'on'

Output assertion signal — Create output signal
off (default) | on

Selecting this check box causes the block to output a Boolean signal that is true (1) at each time step if the assertion succeeds and false (0) if the assertion fails. The data type of the output signal is Boolean if you have selected the Implement logic signals as Boolean data check box on the Configuration Parameters dialog box. Otherwise the data type of the output signal is double.

Command-Line Information
Parameter: export
Type: character vector
Values: 'on' | 'off'
**Default:** 'off'

**Select icon type — Select icon type**

| graphic (default) | text |

Specify the type of icon used to display this block in a block diagram. The **graphic** option displays a graphical representation of the assertion condition on the icon. The **text** option displays a mathematical expression that represents the assertion condition. If the icon is too small to display the expression, the text icon displays an exclamation point. To see the expression, enlarge the block.

**Command-Line Information**

**Parameter:** icon

**Type:** character vector

**Values:** 'graphic' | 'text'

**Default:** 'graphic'

### Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>double</th>
<th>enumerated</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Extended Capabilities

**C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.
For information about how Simulink Coder generated code handles Model Verification blocks, see “Debug” (Simulink Coder).

**HDL Code Generation**
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

**HDL Architecture**
This block can be used to check the upper bound of a test signal as compared to the input, but it is not included in the generated code.

**PLC Code Generation**
Generate Structured Text code using Simulink® PLC Coder™.

**Fixed-Point Conversion**
Design and simulate fixed-point systems using Fixed-Point Designer™.

**See Also**
Check Dynamic Lower Bound | Check Dynamic Range

*Introduced before R2006a*
Check Input Resolution

Check that input signal has specified resolution

Library: Simulink / Model Verification
HDL Coder / Model Verification

Description

The Check Input Resolution block checks whether the input signal has a specified scalar or vector resolution. If the resolution is a scalar, the input signal must be a multiple of the resolution within a $10e^{-3}$ tolerance. If the resolution is a vector, the input signal must equal an element of the resolution vector. If the verification condition is true, the block does nothing. If not, the block halts the simulation, by default, and displays an error message.

Use the blocks in the Model Verification library to facilitate creation of self-validating models. For example, you can use model verification blocks to test that signals do not exceed specified limits during simulation. When you are satisfied that a model is correct, you can turn error checking off by disabling the verification blocks. You do not have to remove them from the model. If you need to modify a model, you can temporarily turn the verification blocks back on to ensure that your changes do not break the model.

Ports

Input

Port_1 — Input signal

scalar | vector | matrix

Input signal that the block checks the resolution specified by the Resolution parameter.

Data Types: double
Output

Port_1 — Assertion output signal
scalar

Output signal at each time step that is true (1) if the assertion succeeds, and false (0) if the assertion fails. If, in the Configuration Parameters, you select **Implement logic signals as Boolean data**, then the output data type is a Boolean. Otherwise the data type of the signal is a double.

Dependencies

To enable this output port, select the **Output assertion signal** parameter check box.

Data Types: double | Boolean

Parameters

Resolution — Resolution
scalar

Specify the resolution requirement for the input signal.

Command-Line Information
Parameter: resolution
Type: character vector
Values: '1' | real value
Default: '1'

Enable assertion — Enable or disable the check
on (default) | off

Clearing this check box disables the block and causes the model to behave as if the block does not exist. You can set the **Model Verification block enabling** setting in the Configuration Parameters to enable or disable all model verification blocks in a model regardless of the setting of this option.

Command-Line Information
Parameter: enabled
Type: character vector
Values: 'on' | 'off'
**Default:** 'on'

**Simulation callback when assertion fails (optional) — Expression to evaluate when assertion fails**

' ' (default) | MATLAB expression

Specify a MATLAB expression to evaluate when the assertion fails. Because the expression is evaluated in the MATLAB workspace, define all variables used in the expression in that workspace.

**Command-Line Information**

**Parameter:** callback

**Type:** character vector

**Values:** MATLAB expression

**Default:** ' '

**Stop simulation when assertion fails — Halt simulation when check fails**

on (default) | off

Select this check box to indicate that the block halts simulation when the check fails. Clear to indicate that the software displays a warning and continues the simulation.

**Command-Line Information**

**Parameter:** stopWhenAssertionFail

**Type:** character vector

**Values:** 'on' | 'off'

**Default:** 'on'

**Output assertion signal — Create output signal**

off (default) | on

Selecting this check box causes the block to output a Boolean signal that is true (1) at each time step if the assertion succeeds and false (0) if the assertion fails. The data type of the output signal is Boolean if you have selected the **Implement logic signals as Boolean data** check box on the Configuration Parameters dialog box. Otherwise the data type of the output signal is double.

**Command-Line Information**

**Parameter:** export

**Type:** character vector

**Values:** 'on' | 'off'

**Default:** 'off'
Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>double</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>yes</td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
</tr>
</tbody>
</table>

Extended Capabilities

**C/C++ Code Generation**
Generate C and C++ code using Simulink® Coder™.

For information about how Simulink Coder generated code handles Model Verification blocks, see “Debug” (Simulink Coder).

Not recommended for production code.

**HDL Code Generation**
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

**HDL Architecture**
This block can be used to check whether the input signal has a specified scalar or vector resolution during simulation, but it is not included in the generated code.
PLC Code Generation
Generate Structured Text code using Simulink® PLC Coder™.

See Also

Introduced before R2006a
Check Static Gap

Check that gap exists in signal's range of amplitudes

**Library:**  
Simulink / Model Verification  
HDL Coder / Model Verification

**Description**

The Check Static Gap block checks that each element of the input signal is less than (or optionally equal to) a static lower bound or greater than (or optionally equal to) a static upper bound at the current time step. If the verification condition is true, the block does nothing. If not, the block halts the simulation, by default, and displays an error message.

Use the blocks in the Model Verification library to facilitate creation of self-validating models. For example, you can use model verification blocks to test that signals do not exceed specified limits during simulation. When you are satisfied that a model is correct, you can turn error checking off by disabling the verification blocks. You do not have to remove them from the model. If you need to modify a model, you can temporarily turn the verification blocks back on to ensure that your changes do not break the model.

**Ports**

**Input**

**Port_1 — Input**

*scalar | vector | matrix*

Input signal the block checks if the signal value is less than a static lower bound or greater than a static upper bound.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 |  
Boolean | fixed point | enumerated | bus
**Output**

**Port_1 — Assertion output signal**

*scalar*

Output signal at each time step that is true (1) if the assertion succeeds, and false (0) if the assertion fails. If, in the Configuration Parameters, you select **Implement logic signals as Boolean data**, then the output data type is a **Boolean**. Otherwise the data type of the signal is a **double**.

**Dependencies**

To enable this output port, set the **Output assertion signal** parameter check box.

Data Types: double | Boolean

**Parameters**

**Upper bound — Upper boundary value**

*scalar | vector | matrix*

Specify the upper bound on the range of amplitudes that the input signal can have.

**Command-Line Information**

**Parameter:** `max`  
**Type:** character vector  
**Values:** scalar | vector | matrix  
**Default:** '0'

**Inclusive upper bound — Include the upper bound in range**

*on (default) | off*

Select this check box to make the range of valid input amplitudes include the upper bound.

**Command-Line Information**

**Parameter:** `max_included`  
**Type:** character vector  
**Values:** 'on' | 'off'  
**Default:** 'on'
**Lower bound — Lower boundary value**
scalar | vector | matrix

Specify the lower bound on the range of amplitudes that the input signal can have.

**Command-Line Information**

**Parameter:** min  
**Type:** character vector  
**Values:** scalar | vector | matrix  
**Default:** '0'

**Inclusive lower bound — Include the lower bound in range**  
on (default) | off

Select this check box to make the range of valid input amplitudes include the lower bound.

**Command-Line Information**

**Parameter:** min_included  
**Type:** character vector  
**Values:** 'on' | 'off'  
**Default:** 'on'

**Enable assertion — Enable or disable the check**  
on (default) | off

Clearing this check box disables the block and causes the model to behave as if the block does not exist. You can set the **Model Verification block enabling** setting in the Configuration Parameters to enable or disable all model verification blocks in a model regardless of the setting of this option.

**Command-Line Information**

**Parameter:** enabled  
**Type:** character vector  
**Values:** 'on' | 'off'  
**Default:** 'on'

**Simulation callback when assertion fails (optional) — Expression to evaluate when assertion fails**  
' ' (default) | MATLAB expression
Specify a MATLAB expression to evaluate when the assertion fails. Because the expression is evaluated in the MATLAB workspace, define all variables used in the expression in that workspace.

**Command-Line Information**
**Parameter:** callback  
**Type:** character vector  
**Values:** MATLAB expression  
**Default:** ''

**Stop simulation when assertion fails — Halt simulation when check fails**  
on (default) | off

Select this check box to indicate that the block halts simulation when the check fails. Clear to indicate that the software displays a warning and continues the simulation.

**Command-Line Information**
**Parameter:** stopWhenAssertionFail  
**Type:** character vector  
**Values:** 'on' | 'off'  
**Default:** 'on'

**Output assertion signal — Create output signal**  
off (default) | on

Selecting this check box causes the block to output a Boolean signal that is true (1) at each time step if the assertion succeeds and false (0) if the assertion fails. The data type of the output signal is Boolean if you have selected the **Implement logic signals as Boolean data** check box on the Configuration Parameters dialog box. Otherwise the data type of the output signal is double.

**Command-Line Information**
**Parameter:** export  
**Type:** character vector  
**Values:** 'on' | 'off'  
**Default:** 'off'

**Select icon type — Select icon type**  
graphic (default) | text

Specify the type of icon used to display this block in a block diagram. The graphic option displays a graphical representation of the assertion condition on the icon. The text option displays a mathematical expression that represents the assertion condition. If the
icon is too small to display the expression, the text icon displays an exclamation point. To see the expression, enlarge the block.

**Command-Line Information**

- **Parameter:** icon
- **Type:** character vector
- **Values:** 'graphic' | 'text'
- **Default:** 'graphic'

**Block Characteristics**

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>double</th>
<th>enumerated</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Extended Capabilities**

**C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

For information about how Simulink Coder generated code handles Model Verification blocks, see “Debug” (Simulink Coder).

Not recommended for production code.
**HDL Code Generation**
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

**HDL Architecture**
This block checks whether each element is less than a static lower bound or greater than a static upper bound during simulation, but it is not included in the generated code.

**PLC Code Generation**
Generate Structured Text code using Simulink® PLC Coder™.

**Fixed-Point Conversion**
Design and simulate fixed-point systems using Fixed-Point Designer™.

**See Also**
Check Dynamic Range | Check Static Lower Bound | Check Static Upper Bound

*Introduced before R2006a*
Check Static Lower Bound

Check that signal is greater than (or optionally equal to) static lower bound

**Library:** Simulink / Model Verification
HDL Coder / Model Verification

**Description**

The Check Static Lower Bound block checks that each element of the input signal is greater than (or optionally equal to) a specified lower bound at the current time step. Specify the value of the lower bound with the **Lower bound** parameter. If the verification condition is true, the block does nothing. If not, the block halts the simulation, by default, and displays an error message.

Use the blocks in the Model Verification library to facilitate creation of self-validating models. For example, you can use model verification blocks to test that signals do not exceed specified limits during simulation. When you are satisfied that a model is correct, you can turn error checking off by disabling the verification blocks. You do not have to remove them from the model. If you need to modify a model, you can temporarily turn the verification blocks back on to ensure that your changes do not break the model.

**Ports**

**Input**

**Port_1 — Input signal**

scalar | vector | matrix

Input signal checked against the lower bound specified by the **Lower bound** parameter.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated
Output

Port_1 — Assertion output signal
scalar

Output signal at each time step that is true (1) if the assertion succeeds, and false (0) if the assertion fails. If, in the Configuration Parameters, you select Implement logic signals as Boolean data, then the output data type is a Boolean. Otherwise the data type of the signal is a double.

Dependencies

To enable this output port, select the Output assertion signal parameter check box.

Data Types: double | Boolean

Parameters

Lower bound — Lower boundary value
scalar | vector | matrix

Specify the lower bound on the range of amplitudes that the input signal can have.

Command-Line Information
Parameter: min
Type: character vector
Values: scalar | vector | matrix
Default: '0'

Inclusive boundary — Include the lower bound in range
on (default) | off

Select this check box to make the range of valid input amplitudes include the lower bound.

Command-Line Information
Parameter: min_included
Type: character vector
Values: 'on' | 'off'
Default: 'on'
Enable assertion — Enable or disable the check
on (default) | off

Clearing this check box disables the block and causes the model to behave as if the block does not exist. You can set the Model Verification block enabling setting in the Configuration Parameters to enable or disable all model verification blocks in a model regardless of the setting of this option.

Command-Line Information
Parameter: enabled
Type: character vector
Values: 'on' | 'off'
Default: 'on'

Simulation callback when assertion fails (optional) — Expression to evaluate when assertion fails
' ' (default) | MATLAB expression

Specify a MATLAB expression to evaluate when the assertion fails. Because the expression is evaluated in the MATLAB workspace, define all variables used in the expression in that workspace.

Command-Line Information
Parameter: callback
Type: character vector
Values: MATLAB expression
Default: ''

Stop simulation when assertion fails — Halt simulation when check fails
on (default) | off

Select this check box to indicate that the block halts simulation when the check fails. Clear to indicate that the software displays a warning and continues the simulation.

Command-Line Information
Parameter: stopWhenAssertionFail
Type: character vector
Values: 'on' | 'off'
Default: 'on'

Output assertion signal — Create output signal
off (default) | on
Selecting this check box causes the block to output a Boolean signal that is true (1) at each time step if the assertion succeeds and false (0) if the assertion fails. The data type of the output signal is Boolean if you have selected the **Implement logic signals as Boolean data** check box on the Configuration Parameters dialog box. Otherwise the data type of the output signal is double.

**Command-Line Information**

**Parameter:** export  
**Type:** character vector  
**Values:** 'on' | 'off'  
**Default:** 'off'

**Select icon type — Select icon type**

`graphic` (default) | `text`

Specify the type of icon used to display this block in a block diagram. The *graphic* option displays a graphical representation of the assertion condition on the icon. The *text* option displays a mathematical expression that represents the assertion condition. If the icon is too small to display the expression, the text icon displays an exclamation point. To see the expression, enlarge the block.

**Command-Line Information**

**Parameter:** icon  
**Type:** character vector  
**Values:** 'graphic' | 'text'  
**Default:** 'graphic'

### Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>double</th>
<th>enumerated</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Extended Capabilities

### C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

For information about how Simulink Coder generated code handles Model Verification blocks, see “Debug” (Simulink Coder).

Not recommended for production code.

### HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

#### HDL Architecture
This block checks whether each input signal element is greater than a static lower bound during simulation, but it is not included in the generated code.

### PLC Code Generation
Generate Structured Text code using Simulink® PLC Coder™.

### Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.
See Also
Check Dynamic Lower Bound | Check Dynamic Range | Check Dynamic Upper Bound | Check Static Upper Bound

Introduced before R2006a
Check Static Range

Check that signal falls inside fixed range of amplitudes

Library:  Simulink / Model Verification
          HDL Coder / Model Verification

Description

The Check Static Range block checks that each element of the input signal falls inside the same range of amplitudes at each time step. Specify the upper and lower bounds with the parameters **Upper bound** and **Lower bound**. If the verification condition is true, the block does nothing. If not, the block halts the simulation, by default, and displays an error message.

Use the blocks in the Model Verification library to facilitate creation of self-validating models. For example, you can use model verification blocks to test that signals do not exceed specified limits during simulation. When you are satisfied that a model is correct, you can turn error checking off by disabling the verification blocks. You do not have to remove them from the model. If you need to modify a model, you can temporarily turn the verification blocks back on to ensure that your changes do not break the model.

Ports

Input

**Port_1 — Input signal**

*scalar | vector | matrix*

Input signal checked against the range specified by the **Upper bound** and **Lower bound** parameters.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated
Output

Port_1 — Assertion output signal
scalar

Output signal at each time step that is true (1) if the assertion succeeds, and false (0) if
the assertion fails. If, in the Configuration Parameters, you select Implement logic
signals as Boolean data, then the output data type is a Boolean. Otherwise the data
type of the signal is a double.

Dependencies

To enable this output port, set the Output assertion signal parameter check box.
Data Types: double | Boolean

Parameters

Upper bound — Upper boundary value
scalar | vector | matrix

Specify the upper bound on the range of amplitudes that the input signal can have.

Command-Line Information
Parameter: max
Type: character vector
Values: scalar | vector | matrix
Default: '0'

Inclusive upper boundary — Include the upper bound in range
on (default) | off

Select this check box to make the range of valid input amplitudes include the lower
bound.

Command-Line Information
Parameter: min_included
Type: character vector
Values: 'on' | 'off'
Default: 'on'
**Lower bound** — Lower boundary value
scalar | vector | matrix

Specify the lower bound on the range of amplitudes that the input signal can have.

**Command-Line Information**
**Parameter:** min
**Type:** character vector
**Values:** scalar | vector | matrix
**Default:** '0'

**Inclusive lower bound** — Include the lower bound in range
on (default) | off

Select this check box to make the range of valid input amplitudes include the lower bound.

**Command-Line Information**
**Parameter:** min_included
**Type:** character vector
**Values:** 'on' | 'off'
**Default:** 'on'

**Enable assertion** — Enable or disable the check
on (default) | off

Clearing this check box disables the block and causes the model to behave as if the block does not exist. You can set the Model Verification block enabling setting in the Configuration Parameters to enable or disable all model verification blocks in a model regardless of the setting of this option.

**Command-Line Information**
**Parameter:** enabled
**Type:** character vector
**Values:** 'on' | 'off'
**Default:** 'on'

**Simulation callback when assertion fails (optional)** — Expression to evaluate when assertion fails
' ' (default) | MATLAB expression
Specify a MATLAB expression to evaluate when the assertion fails. Because the expression is evaluated in the MATLAB workspace, define all variables used in the expression in that workspace.

**Command-Line Information**

**Parameter:** callback  
**Type:** character vector  
**Values:** MATLAB expression  
**Default:** ''

**Stop simulation when assertion fails — Halt simulation when check fails**

on (default) | off

Select this check box to indicate that the block halts simulation when the check fails. Clear to indicate that the software displays a warning and continues the simulation.

**Command-Line Information**

**Parameter:** stopWhenAssertionFail  
**Type:** character vector  
**Values:** 'on' | 'off'  
**Default:** 'on'

**Output assertion signal — Create output signal**

off (default) | on

Selecting this check box causes the block to output a Boolean signal that is true (1) at each time step if the assertion succeeds and false (0) if the assertion fails. The data type of the output signal is Boolean if you have selected the **Implement logic signals as Boolean data** check box on the Configuration Parameters dialog box. Otherwise the data type of the output signal is double.

**Command-Line Information**

**Parameter:** export  
**Type:** character vector  
**Values:** 'on' | 'off'  
**Default:** 'off'

**Select icon type — Select icon type**

graphic (default) | text

Specify the type of icon used to display this block in a block diagram. The graphic option displays a graphical representation of the assertion condition on the icon. The text option displays a mathematical expression that represents the assertion condition. If the
icon is too small to display the expression, the text icon displays an exclamation point. To see the expression, enlarge the block.

**Command-Line Information**

**Parameter:** icon  
**Type:** character vector  
**Values:** 'graphic' | 'text'  
**Default:** 'graphic'

### Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>double</th>
<th>enumerated</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Direct Feedthrough</th>
<th>no</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multidimensional Signals</td>
<td>yes</td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
</tr>
</tbody>
</table>

### Extended Capabilities

**C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

For information about how Simulink Coder generated code handles Model Verification blocks, see “Debug” (Simulink Coder).

Not recommended for production code.
HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture
This block checks whether each input signal element falls within the same amplitude range at each time step during simulation, but it is not included in the generated code.

PLC Code Generation
Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

See Also
Check Dynamic Range | Check Static Lower Bound | Check Static Upper Bound

Introduced before R2006a
Check Static Upper Bound

Check that signal is less than (or optionally equal to) static upper bound

Library: Simulink / Model Verification
HDLCoder / Model Verification

Description

The Check Static Upper Bound block checks that each element of the input signal is less than (or optionally equal to) a specified upper bound at the current time step. Use the block parameter dialog box to specify the value of the upper bound and whether the bound is inclusive. If the verification condition is true, the block does nothing. If not, the block halts the simulation, by default, and displays an error message.

Use the blocks in the Model Verification library to facilitate creation of self-validating models. For example, you can use model verification blocks to test that signals do not exceed specified limits during simulation. When you are satisfied that a model is correct, you can turn error checking off by disabling the verification blocks. You do not have to remove them from the model. If you need to modify a model, you can temporarily turn the verification blocks back on to ensure that your changes do not break the model.

Ports

Input

Port_1 — Input signal
scalar | vector | matrix

Input signal checked against the upper bound specified by the Upper bound parameter.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated
Output

Port_1 — Assertion output signal
scalar

Output signal at each time step that is true (1) if the assertion succeeds, and false (0) if the assertion fails. If, in the Configuration Parameters, you select Implement logic signals as Boolean data, then the output data type is a Boolean. Otherwise the data type of the signal is a double.

Dependencies

To enable this output port, select the Output assertion signal parameter check box.

Data Types: double | Boolean

Parameters

Upper bound — Upper boundary value
scalar | vector | matrix

Specify the upper bound on the range of amplitudes that the input signal can have.

Command-Line Information
Parameter: max
Type: character vector
Values: scalar | vector | matrix
Default: '0'

Inclusive boundary — Include the upper bound in range
on (default) | off

Select this check box to make the range of valid input amplitudes include the upper bound.

Command-Line Information
Parameter: max_included
Type: character vector
Values: 'on' | 'off'
Default: 'on'
Enable assertion — Enable or disable the check
on (default) | off

Clearing this check box disables the block and causes the model to behave as if the block does not exist. You can set the Model Verification block enabling setting in the Configuration Parameters to enable or disable all model verification blocks in a model regardless of the setting of this option.

Command-Line Information
Parameter: enabled
Type: character vector
Values: 'on' | 'off'
Default: 'on'

Simulation callback when assertion fails (optional) — Expression to evaluate when assertion fails
' ' (default) | MATLAB expression

Specify a MATLAB expression to evaluate when the assertion fails. Because the expression is evaluated in the MATLAB workspace, define all variables used in the expression in that workspace.

Command-Line Information
Parameter: callback
Type: character vector
Values: MATLAB expression
Default: '

Stop simulation when assertion fails — Halt simulation when check fails
on (default) | off

Select this check box to indicate that the block halts simulation when the check fails. Clear to indicate that the software displays a warning and continues the simulation.

Command-Line Information
Parameter: stopWhenAssertionFail
Type: character vector
Values: 'on' | 'off'
Default: 'on'

Output assertion signal — Create output signal
off (default) | on
Selecting this check box causes the block to output a Boolean signal that is true (1) at each time step if the assertion succeeds and false (0) if the assertion fails. The data type of the output signal is Boolean if you have selected the Implement logic signals as Boolean data check box on the Configuration Parameters dialog box. Otherwise the data type of the output signal is double.

**Command-Line Information**
**Parameter:** export
**Type:** character vector
**Values:** 'on' | 'off'
**Default:** 'off'

**Select icon type — Select icon type**
**graphic (default) | text**

Specify the type of icon used to display this block in a block diagram. The graphic option displays a graphical representation of the assertion condition on the icon. The text option displays a mathematical expression that represents the assertion condition. If the icon is too small to display the expression, the text icon displays an exclamation point. To see the expression, enlarge the block.

**Command-Line Information**
**Parameter:** icon
**Type:** character vector
**Values:** 'graphic' | 'text'
**Default:** 'graphic'

**Block Characteristics**

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>double</th>
<th>enumerated</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Extended Capabilities

**C/C++ Code Generation**
Generate C and C++ code using Simulink® Coder™.

For information about how Simulink Coder generated code handles Model Verification blocks, see “Debug” (Simulink Coder).

Not recommended for production code.

**HDL Code Generation**
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

**HDL Architecture**
This block checks whether each input signal element is lower than a static upper bound during simulation, but it is not included in the generated code.

**PLC Code Generation**
Generate Structured Text code using Simulink® PLC Coder™.

**Fixed-Point Conversion**
Design and simulate fixed-point systems using Fixed-Point Designer™.
See Also
Check Dynamic Lower Bound | Check Dynamic Range | Check Dynamic Upper Bound | Check Static Lower Bound

Introduced before R2006a
Chirp Signal

Generate sine wave with increasing frequency

Library: Simulink / Sources

Description

The Chirp Signal block generates a sine wave whose frequency increases at a linear rate with time. You can use this block for spectral analysis of nonlinear systems. The block generates a scalar or vector output.

The parameters, Initial frequency, Target time, and Frequency at target time, determine the block’s output. You can specify any or all of these variables as scalars or arrays. All the parameters specified as arrays must have the same dimensions. The block expands scalar parameters to have the same dimensions as the array parameters. The block output has the same dimensions as the parameters unless you select the Interpret vector parameters as 1-D check box. If you select this check box and the parameters are row or column vectors, the block outputs a vector (1-D array) signal.

Limitations

- The start time of the simulation must be 0. To confirm this value, go to the Solver pane in the Configuration Parameters dialog box and view the Start time field.
- Suppose that you use a Chirp Signal block in an enabled subsystem. Whenever the subsystem is enabled, the block output matches what would appear if the subsystem were enabled throughout the simulation.
Ports

Output

Port 1 — Chirp signal
scalar | vector | matrix | N-D array

Sine wave whose frequency increases at a linear rate with time. The chirp signal can be a scalar, vector, matrix, or N-D array.

Data Types: double

Parameters

Initial frequency — Initial frequency (Hz)
0.1 (default) | scalar | vector | matrix | N-D array

The initial frequency of the signal, specified as a scalar, vector, matrix, or N-D array.

Programmatic Use
Block Parameter: f1
Type: character vector
Values: scalar | vector | matrix | N-D array
Default: '0.1'

Target time — Target time (seconds)
100 (default) | scalar | vector | matrix | N-D array

Time, in seconds, at which the frequency reaches the Frequency at target time parameter value. You specify the Target time as a scalar, vector, matrix, or N-D array. After the target time is reached, the frequency continues to change at the same rate.

Programmatic Use
Block Parameter: T
Type: character vector
Values: scalar | vector | matrix | N-D array
Default: '100'

Frequency at target time — Frequency (Hz)
1 (default) | scalar | vector | matrix | N-D array
Frequency, in Hz, of the signal at the target time, specified as a scalar, vector, matrix, or N-D array.

**Programmatic Use**

**Block Parameter:** f2  
**Type:** character vector  
**Values:** scalar | vector | matrix | N-D array  
**Default:** '1'

**Interpret vector parameters as 1-D — Treat vector parameters as 1-D**

When you select this check box, any column or row matrix values you specify for the Initial frequency, Target time, and Frequency at target time parameters result in a vector output whose elements are the elements of the row or column. For more information, see “Determining the Output Dimensions of Source Blocks”.

**Programmatic Use**

**Block Parameter:** VectorParams1D  
**Type:** character vector  
**Values:** 'on' | 'off'  
**Default:** 'on'

**Block Characteristics**

<table>
<thead>
<tr>
<th>Data Types</th>
<th>double</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>no</td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
</tr>
</tbody>
</table>
Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

Not recommended for production code. Relates to resource limits and restrictions on speed and memory often found in embedded systems. Generated code can contain dynamic allocation and freeing of memory, recursion, additional memory overhead, and widely-varying execution times. While the code is functionally valid and generally acceptable in resource-rich environments, smaller embedded targets often cannot support such code. Usually, blocks evolve toward being suitable for production code. Thus, blocks suitable for production code remain suitable.

See Also
Sine Wave

Topics
“Creating Signals”

Introduced before R2006a
Clock

Display and provide simulation time
Library: Simulink / Sources

Description

The Clock block outputs the current simulation time at each simulation step. This block is useful for other blocks that need the simulation time.

When you need the current time within a discrete system, use the Digital Clock block.

Ports

Output

Port_1 — Sample time
scalar

Sample time, specified as the current simulation time at each simulation time step.

Data Types: double

Parameters

Display time — Display simulation time on block icon
off (default) | on

Select this check box to display the simulation time as part of the Clock block icon. When you clear this check box, the simulation time does not appear on the block icon.
Programmatic Use
Block Parameter: DisplayTime
Type: character vector
Values: 'off' | 'on'
Default: 'off'

Decimation — Interval at which to update block icon
10 (default) | positive integer

Specify the interval at which Simulink updates the Clock icon as a positive integer.
Suppose that the decimation is 1000. For a fixed integration step of 1 millisecond, the Clock icon updates at 1 second, 2 seconds, and so on.

Dependencies
To display the simulation time on the block icon, you must select the Display time check box.

Programmatic Use
Block Parameter: Decimation
Type: character vector
Value: scalar
Default: '10'

Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>double</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>no</td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
</tr>
</tbody>
</table>
**Extended Capabilities**

**C/C++ Code Generation**
Generate C and C++ code using Simulink® Coder™.

Not recommended for production code. Relates to resource limits and restrictions on speed and memory often found in embedded systems. Generated code can contain dynamic allocation and freeing of memory, recursion, additional memory overhead, and widely-varying execution times. While the code is functionally valid and generally acceptable in resource-rich environments, smaller embedded targets often cannot support such code. Usually, blocks evolve toward being suitable for production code. Thus, blocks suitable for production code remain suitable.

**See Also**
Digital Clock

**Topics**
“Sample Time”

**Introduced before R2006a**
Combinatorial Logic

Implement truth table
Library: Simulink / Logic and Bit Operations

Description

The Combinatorial Logic block implements a standard truth table for modeling programmable logic arrays (PLAs), logic circuits, decision tables, and other Boolean expressions. You can use this block in conjunction with Memory blocks to implement finite-state machines or flip-flops.

Ports

Input

Port_1 — Input signal
vector

Input signal, specified as a vector. The type of signals accepted by a Combinatorial Logic block depends on whether you selected the Boolean logic signals option (see “Implement logic signals as Boolean data (vs. double)”). If this option is enabled, the block accepts real signals of type Boolean or double.

Data Types: double | Boolean

Output

Port_2 — Output signal
scalar | vector
Output signal, double if the truth table contains non-Boolean values of type double; Boolean otherwise. The type of the output is the same as that of the input except that the block outputs double if the input is Boolean and the truth table contains non-Boolean values.

Data Types: double | Boolean

Parameters

Truth table — Matrix of outputs

matrix

You specify a matrix that defines all possible block outputs as the Truth table parameter. Each row of the matrix contains the output for a different combination of input elements. You must specify outputs for every combination of inputs. The number of columns is the number of block outputs.

The Truth table parameter can have Boolean values (0 or 1) of any data type, including fixed-point data types. If the table contains non-Boolean values, the data type of the table must be double.

The relationship between the number of inputs and the number of rows is:

number of rows = 2^(number of inputs)

Simulink returns a row of the matrix by computing the row's index from the input vector elements. Simulink computes the index by building a binary number where input vector elements having zero values are 0 and elements having nonzero values are 1, then adding 1 to the result. For an input vector, u, of m elements:

row index = 1 + u(m)*2^0 + u(m-1)*2^1 + ... + u(1)*2^m-1

Programmatic Use

Block Parameter: TruthTable
Type: character vector
Values: matrix
Default: ' [0 0; 0 1; 0 1; 0 1; 0 1; 0 1 1]'
Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>double</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
</tr>
</tbody>
</table>

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also
Bit Clear | Bit Set | Compare To Constant | Compare To Zero

Introduced before R2006a
Combo Box

Select parameter value from drop-down menu

**Library:** Simulink / Dashboard

**Description**

The Combo Box block lets you set the value of a parameter to one of several values. You can define each selectable value and its label through the Combo Box block parameters. Use the Combo Box block with other Dashboard blocks to build an interactive dashboard of controls and indicators for your model.

Double-clicking the Combo Box block does not open its dialog box during simulation and when the block is selected. To edit the block's parameters, you can use the **Property Inspector**, or you can right-click the block and select **Block Parameters** from the context menu.

**Connecting Dashboard Blocks**

Dashboard blocks do not use ports to make connections. To connect Dashboard blocks to variables and block parameters in your model, use connect mode. Connect mode facilitates the process of connecting Dashboard blocks in your model, especially when you want to connect multiple blocks at once. If you only want to connect a single Dashboard block, you can also use the **Connection** table in the block dialog box to make the connection.

**Tip** To facilitate understanding and debugging your model, you can modify Dashboard block connections in your model during normal and accelerator mode simulations.

**Note** Dashboard blocks cannot connect to variables until you update your model diagram. To connect Dashboard blocks to variables or modify variable values between opening your model and running a simulation, update your model diagram using **Ctrl+D**.
To enter connect mode, click the **Connect** button that appears above your unconnected Dashboard block when you pause on it.

In connect mode, when you select one or more signals or blocks, a list of parameters and signals available for connection appears. Select a signal or parameter from the list to connect the selected Dashboard block. To connect another Dashboard block, pause on the block and click the **Connect** button above it. Then, make a selection of signals and blocks in your model, and choose a signal or parameter to connect.
When you finish connecting the Dashboard blocks in your model, click the **Exit** button in the upper-right of the canvas to exit connect mode.

**Tip** You can hide the message shown on unconnected blocks using the `set_param` function with the `ShowInitialText` block parameter.

### Parameter Logging

Tunable parameters connected to Dashboard blocks are logged to the Simulation Data Inspector, where you can view parameter values along with logged signal data. You can access logged parameter data in the MATLAB workspace by exporting the parameter data from the Simulation Data Inspector UI or by using the `Simulink.sdi.exportRun` function. For more information about exporting data with the Simulation Data Inspector UI, see “Export Data from the Simulation Data Inspector”. The parameter data is stored in a `Simulink.SimulationData.Parameter` object, accessible as an element in the exported `Simulink.SimulationData.Dataset`. 
Limitations

- Dashboard blocks can only connect to real scalar signals.
- You cannot use the Connection table to connect a Dashboard block to a block that is commented out. When you connect a Dashboard block to a commented block using connect mode, the Dashboard block does not display the connected value until you uncomment the block.
- Dashboard blocks cannot connect to signals inside referenced models.
- When you simulate a model hierarchy, Dashboard blocks inside referenced models do not update.
- Dashboard blocks do not support rapid accelerator simulation.
- Some signals are not available during accelerator mode simulations. To view such signals on a Dashboard block, mark the signal for logging.
- Parameters specified by indexing a variable array do not appear in the Connection table. For example, a block parameter defined as engine(1) using the variable engine does not appear in the table.

To access the parameter in the Connection table, assign the indexed value to a scalar variable, such as engine_1. Then, use the scalar variable to define the block parameter.

- When you connect a Dashboard block to a variable or parameter during simulation, the data for that variable or parameter does not get logged to the Simulation Data Inspector. To log variable and parameter data to the Simulation Data Inspector, connect the Dashboard block to the variable or parameter prior to simulation.
- When you simulate a model in external mode with the Default parameter behavior set to Inlined, Dashboard blocks can appear to change parameter and variable values. However, the change does not propagate to the simulation. For example, the Gain block displays changes made to its Gain parameter using the Dashboard blocks, but the Gain value used in the simulation does not change.

Parameters

Connection — Select a variable or block parameter to connect
variable and parameter connection options

Select the variable or block parameter to control using the Connection table. Populate the Connection table by selecting one or more blocks in your model. Select the radio
button next to the variable or parameter you want to control, and click Apply. To facilitate understanding and debugging your model, you can connect Dashboard blocks to variables and parameters in your model during simulation.

**Note** To see workspace variables in the connection table, update the model diagram using Ctrl+D.

### Programmatic Use

To programatically connect a Dashboard block to a tunable parameter or a variable, use a Simulink.HMI.ParamSourceInfo object. The Simulink.HMI.ParamSourceInfo object contains four properties. Some of the properties apply to connecting Dashboard blocks to parameters, and some apply to connecting Dashboard blocks to variables. Not all fields have a value for a connection because a given Dashboard block connects to either a parameter or a variable.

**Block Parameter:** Binding

**Type:** Simulink.HMI.ParamSourceInfo

**Default:** []

**States** — Set of states to select for connected parameter

Set of states from which to select a value for the connected parameter. Each State consists of a **State Value** and a **State Label**.

- **State Value** — Value assigned to the connected variable or parameter when you select the state with the corresponding **Label**.
- **State Label** — Label for each state. You can use the **Label** to display the value the connected parameter takes when the switch is positioned at the bottom, or you can enter a descriptive text label.

Click the + button to add additional **States**.

The default configuration for the block includes these **States**.
States

<table>
<thead>
<tr>
<th>State Value</th>
<th>State Label</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Label1</td>
</tr>
<tr>
<td>1</td>
<td>Label2</td>
</tr>
<tr>
<td>2</td>
<td>Label3</td>
</tr>
</tbody>
</table>

**Programmatic Use**

To programmatically configure the States for a block, use an array of structures containing the fields `Value` and `Label`. Include a structure in the array for each state you want to configure on the block.

```matlab
state1.Value = 1;
state1.Label = 'State 1';
state2.Value = 2;
state2.Label = 'State 2';
radioStates = [state1 state2];
```

**Block Parameter:** States  
**Type:** structure  
**Default:** 3x1 structure array

**Enumerated Data Type — Specify state values and labels using an enumerated data type**

off (default) | on

You can use an enumerated data type that pairs a numeric value with each enumeration to configure the state values and labels for the block. To specify the states for the block using an enumerated data type, first select the **Enumerated Data Type** option. Then, specify the name of the enumerated data type in the text box. The definition for the specified enumerated data type must be saved on the MATLAB path or in the base workspace.

Example: `myEnumType`

**Programmatic Use**

To programmatically specify the state labels and values for the block using an enumerated data type, specify 'on' for the **UseEnumeratedDataType** parameter and the name of the enumerated data type for the **EnumeratedDataType** parameter.  
**Block Parameter:** **UseEnumeratedDataType**
Type: string or character array
Values: 'on' | 'off'
Default: 'off'
Block Parameter: EnumeratedDataType
Type: string or character array
Default: ''

Label — Block label position
Hide (default) | Bottom | Top

Position of the block label. When the block is connected to an element in the model, the label is the name of the connected element.

Programmatic Use
Block Parameter: LabelPosition
Type: character vector
Values: 'Hide' | 'Bottom' | 'Top'
Default: 'Hide'

Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>double</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>no</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.
Dashboard blocks are ignored for code generation.

See Also
Radio Button | Rotary Switch

Topics
“Tune and Visualize Your Model with Dashboard Blocks”
“Decide How to Visualize Simulation Data”

Introduced in R2017b
Compare To Constant

Determine how signal compares to specified constant

Library: Simulink / Logic and Bit Operations
        HDL Coder / Logic and Bit Operations

Description

The Compare To Constant block compares an input signal to a constant. Specify the constant in the Constant value parameter. Specify how the input is compared to the constant value with the Operator parameter.

Ports

Input

Port_1 — Input signal
scalar

Input signal, specified as a scalar, is compared with zero. The block first converts its Constant value parameter to the input data type, and then performs the specified operation.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus

Output

Port_1 — Output signal
0 | 1

The output is 0 if the comparison is false, and 1 if it is true.
Data Types: uint8 | Boolean

Parameters

Operator — Logical operator
<= (default) | == | ~= | < | >= | >

This parameter can have these values:

- == — Determine whether the input is equal to the specified constant.
- ~= — Determine whether the input is not equal to the specified constant.
- < — Determine whether the input is less than the specified constant.
- <= — Determine whether the input is less than or equal to the specified constant.
- > — Determine whether the input is greater than the specified constant.
- >= — Determine whether the input is greater than or equal to the specified constant.

Programmatic Use
Block Parameter: relop
Type: character vector
Values: '==' | '~=' | '<' | '<=' | '>=' | '>'
Default: '<='

Constant value — Constant to compare with
constant

Specify the constant value to which the input is compared.

Programmatic Use
Block Parameter: const
Type: character vector
Value: scalar | vector | matrix | N-D array
Default: '3.0'

Output data type — Data type of the output
boolean (default) | uint8

Specify the data type of the output, boolean or uint8.
Programmatic Use
Block Parameter: OutDataTypeStr
Type: character vector
Values: 'boolean' | 'uint8'
Default: 'boolean'

Enable zero-crossing detection — Select to enable zero-crossing detection check (default) | uncheck

Select to enable zero-crossing detection. For more information, see “Zero-Crossing Detection” in the Simulink documentation.

Programmatic Use
Block Parameter: ZeroCross
Type: character vector
Values: 'off' | 'on'
Default: 'on'

Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>double</th>
<th>enumerated</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture
This block has a single, default HDL architecture.

HDL Block Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConstrainedOutputPipeline</td>
<td>Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. For more details, see “ConstrainedOutputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td>InputPipeline</td>
<td>Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “InputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td>OutputPipeline</td>
<td>Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “OutputPipeline” (HDL Coder).</td>
</tr>
</tbody>
</table>

Complex Data Support
This block supports code generation for complex signals.

PLC Code Generation
Generate Structured Text code using Simulink® PLC Coder™.
Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

See Also
Combinatorial Logic | Compare To Zero | Logical Operator

Introduced before R2006a
Compare To Zero

Determine how signal compares to zero

**Library:**
- Simulink / Logic and Bit Operations
- HDL Coder / Logic and Bit Operations

**Description**

The Compare To Zero block compares an input signal to zero. Specify how the input is compared to zero with the **Operator** parameter.

The output is 0 if the comparison is false, and 1 if it is true.

**Ports**

**Input**

**Port_1 — Input signal**
- scalar

Input signal, specified as scalar, is compared with zero. If the input data type cannot represent zero, parameter overflow occurs. To detect this overflow, go to the **Diagnostics > Data Validity** pane of the Configuration Parameters dialog box and set **Parameters > Detect overflow** to warning or error.

In this case, the block compares the input signal to the ground value of the input data type. For example, if you have an input signal of type `fixdt(0, 8, 2^0, 10)`, the input data type can represent unsigned 8-bit integers from 10 to 265 due to a bias of 10. The ground value is 10, instead of 0.

**Data Types:** single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point
Output

Port_1 — Output signal
0 | 1

The output is 0 if the comparison is false, and 1 if it is true.

The block output is uint8 or boolean, depending on your selection for the Output data type parameter.

Data Types: uint8 | Boolean

Parameters

Operator — Logical operator
< (default) | == | ~= | < | >= | >

This parameter can have the following values:

- == — Determine whether the input is equal to zero.
- ~= — Determine whether the input is not equal to zero.
- < — Determine whether the input is less than zero.
- <= — Determine whether the input is less than or equal to zero.
- > — Determine whether the input is greater than zero.
- >= — Determine whether the input is greater than or equal to zero.

Programmatic Use
Block Parameter: relop
Type: character vector
Values: '==' | '~=' | '<' | '<=' | '>' | '=>' | '>
Default: '<='

Output data type — Data type of the output
boolean (default) | uint8

Specify the data type of the output, boolean or uint8.

Programmatic Use
Block Parameter: OutDataTypeStr
**Type**: character vector  
**Values**: 'boolean' | 'uint8'  
**Default**: 'boolean'

**Enable zero-crossing detection** — Select to enable zero-crossing detection  
on (default) | off

Select to enable zero-crossing detection. For more information, see “Zero-Crossing Detection” in the Simulink documentation.

**Programmatic Use**  
**Block Parameter**: ZeroCross  
**Type**: character vector  
**Values**: 'off' | 'on'  
**Default**: 'on'

### Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>double</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Extended Capabilities

**C/C++ Code Generation**  
Generate C and C++ code using Simulink® Coder™.
**HDL Code Generation**
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

**HDL Architecture**
This block has a single, default HDL architecture.

**HDL Block Properties**

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConstrainedOutputPipeline</td>
<td>Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. For more details, see “ConstrainedOutputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td>InputPipeline</td>
<td>Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “InputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td>OutputPipeline</td>
<td>Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “OutputPipeline” (HDL Coder).</td>
</tr>
</tbody>
</table>

**Complex Data Support**
This block supports code generation for complex signals.

**PLC Code Generation**
Generate Structured Text code using Simulink® PLC Coder™.

**Fixed-Point Conversion**
Design and simulate fixed-point systems using Fixed-Point Designer™.
See Also
Bitwise Operator | Compare To Constant | Logical Operator | String Compare

Introduced before R2006a
Complex to Magnitude-Angle

Compute magnitude and/or phase angle of complex signal

Library: Simulink / Math Operations

Description

The Complex to Magnitude-Angle block outputs the magnitude and/or phase angle of the input signal, depending on the setting of the Output parameter. The outputs are real values of the same data type as the block input. The input can be an array of complex signals, in which case the output signals are also arrays. The magnitude signal array contains the magnitudes of the corresponding complex input elements. The angle output similarly contains the angles of the input elements.

Ports

Input

Port_1 — Input signal
scalar | vector | matrix

Complex input signal that the block computes and outputs the magnitude and/or the phase angle.

Data Types: single | double

Output

Port_1 — Output signal
scalar | vector | matrix
Output signal that is the magnitude and/or phase angle of the input signal. To choose the output, set the **Output** parameter.

Data Types: single | double

### Parameters

**Output** — Magnitude and/or phase angle output specification  
Magnitude and angle (default) | Magnitude | Angle

Specify if the output is the magnitude and/or the phase angle in radians of the input signal.

**Command-Line Information**

Parameter: Output  
Type: character vector  
Values: 'Magnitude and angle' | 'Magnitude' | 'Angle'  
Default: 'Magnitude and angle'

**Sample time** — Specify sample time as a value other than -1  
-1 (default) | scalar | vector

Specify the sample time as a value other than -1. For more information, see “Specify Sample Time”.

### Dependencies

This parameter is not visible unless it is explicitly set to a value other than -1. To learn more, see “Blocks for Which Sample Time Is Not Recommended”.

**Programmatic Use**

Block Parameter: SampleTime  
Type: character vector  
Values: scalar or vector  
Default: '-1'

### Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>double</th>
<th>single</th>
</tr>
</thead>
</table>

1-208
Complex to Magnitude-Angle

<table>
<thead>
<tr>
<th>Feature</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>yes</td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>yes</td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>yes</td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
</tr>
</tbody>
</table>

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also
Complex to Real-Imag | Real-Imag to Complex

Introduced before R2006a
Complex to Real-Imag

Output real and imaginary parts of complex input signal

Library:
Simulink / Math Operations
HDL Coder / Math Operations

Description

The Complex to Real-Imag block outputs the real and/or imaginary part of the input signal, depending on the setting of the Output parameter. The real outputs are of the same data type as the complex input. The input can be an array (vector or matrix) of complex signals, in which case the output signals are arrays of the same dimensions. The real array contains the real parts of the corresponding complex input elements. The imaginary output similarly contains the imaginary parts of the input elements.

Ports

Input

Port_1 — Input signal
scalar | vector | matrix

Complex input signal that the block computes and outputs the real and/or imaginary part.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point

Output

Port_1 — Output signal
scalar | vector | matrix
Output signal that is the real and/or imaginary part of the input signal. To choose which part is output, set the **Output** parameter.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point

## Parameters

**Output — Real and/or imaginary output specification**

Real and imag (default) | Real | Imag

Specify if the output is the real and/or imaginary part of the input signal.

### Command-Line Information

**Parameter:** Output  
**Type:** character vector  
**Values:** 'Real and imag' | 'Real' | 'Imag'  
**Default:** 'Real and imag'

**Sample time — Specify sample time as a value other than -1**

-1 (default) | scalar | vector

Specify the sample time as a value other than -1. For more information, see “Specify Sample Time”.

### Dependencies

This parameter is not visible unless it is explicitly set to a value other than -1. To learn more, see “Blocks for Which Sample Time Is Not Recommended”.

### Programmatic Use

**Block Parameter:** SampleTime  
**Type:** character vector  
**Values:** scalar or vector  
**Default:** '-1'

## Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>double</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
</table>

1-211
<table>
<thead>
<tr>
<th>Feature</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>yes</td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>yes</td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>yes</td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
</tr>
</tbody>
</table>

## Extended Capabilities

### C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

### HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

#### HDL Architecture
This block has a single, default HDL architecture.

#### HDL Block Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConstrainedOutputPipeline</td>
<td>Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. For more details, see “ConstrainedOutputPipeline” (HDL Coder).</td>
</tr>
</tbody>
</table>
### InputPipeline
Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “InputPipeline” (HDL Coder).

### OutputPipeline
Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “OutputPipeline” (HDL Coder).

#### Complex Data Support
This block supports code generation for complex signals.

#### Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

#### See Also
Complex to Magnitude-Angle | Real-Imag to Complex

**Introduced before R2006a**
Compose String

Compose output string signal based on specified format and input signals
Library: Simulink / String

Description

The Compose String block composes output string signal based on the format specifier listed in the Format parameter. The Format parameter determines the number of input signals. If there are multiple inputs, the block constructs the string by combining these multiple inputs in order, and applying the associated format specifier, one format specifier for each input. Each format specifier starts with a percent sign, %, followed by the conversion character. For example, %f formats the input as a floating point output. To supplement the string output, you can also add a character to the format specification. Use this block to compose and format an output string signal from a multiple inputs.

For example, if the Format parameter contains "%s is %f", the block expects two inputs, a string signal and a single or double signal. If the first input is the string "Pi" and the second input is a double value 3.14, the output is "Pi is 3.14".

When a MinGW® compiler compiles code generated from the block, running the compiled code may produce nonstandard results for floating-point inputs. For example, a numeric input of 501.987 returns the string "5.019870e+002" instead of the expected string "5.019870e+02".

Ports

Input

d — Data for first part of string
scalar
Data for the first part of string, specified as a scalar. The **Format** parameter determines the port label and the format of the input data. For example, if the first item in the **Format** parameter is `%d`, the port label is `d`.

The data type of the input signal must be compatible with the format specifier in the **Format** parameter. For more information, see the **Format** parameter.

**Data Types:** single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean

`f` — Data for second part of string

scalar

Data for the second part of string, specified as a scalar. The **Format** parameter determines the port label and the format of the input data. For example, if the first item in the **Format** parameter is `%f`, the port label is `f`.

The data type of the input signal must be compatible with the format specifier specified in the **Format** parameter. For more information, see the **Format** parameter.

**Data Types:** single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus

**Port_N** — Data for N parts of string

scalar (default)

Data for `N` parts of string, specified as a scalar. The **Format** parameter determines the port label and the format of the input data. For example, if the corresponding item in the **Format** parameter is `%f`, the port label is `f`.

The data type of the input signal must be compatible with the format specifier in the **Format** parameter. For more information, see the **Format** parameter.

**Data Types:** single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus

**Output**

**str** — Output string

scalar

Output string composed of inputs, specified as a scalar.
Data Types: string

**Parameters**

**Format** — Format input data
"%d %f" (default) | scalar

Format of input data, specified as a scalar.

For more information about acceptable format specifiers, see the Algorithms section.

**Output data type** — Output data type
string (default) | scalar

Output data type, specified using the string data type to specify a string with no maximum length.

To specify a string data type with a maximum length, specify `stringtype(N)`. For example, `stringtype(31)` creates a string data type with a maximum length of 31 characters.

Click the **Show data type assistant** button to display the Data Type Assistant, which helps you set the data type attributes. See “Specify Data Types Using Data Type Assistant” in the *Simulink User's Guide* for more information.

**Block Characteristics**

<table>
<thead>
<tr>
<th>Data Types</th>
<th>double</th>
<th>integer</th>
<th>single</th>
<th>string</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Algorithms

A formatting specifier starts with a percent sign, %, and ends with a conversion character. The conversion character is required. Optionally, you can specify identifier, flags, field width, precision, and subtype specifiers between % and the conversion character. (Spaces are invalid between specifiers and are shown here only for readability).

The Compose String block uses this format specifier prototype:

%[flags][width][.precision][length]specifier

Conversion Character

This table shows conversion characters to format numeric and character data as text.

<table>
<thead>
<tr>
<th>Value Type</th>
<th>Conversion</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer, signed</td>
<td>%d or %i</td>
<td>Base 10</td>
</tr>
<tr>
<td>Integer, unsigned</td>
<td>%u</td>
<td>Base 10</td>
</tr>
<tr>
<td></td>
<td>%o</td>
<td>Base 8 (octal)</td>
</tr>
<tr>
<td></td>
<td>%x</td>
<td>Base 16 (hexadecimal), lowercase letters a–f</td>
</tr>
<tr>
<td></td>
<td>%X</td>
<td>Same as %x, uppercase letters A–F</td>
</tr>
<tr>
<td>Floating-point number</td>
<td>%f</td>
<td>Floating-point notation (Use a precision operator to specify the number of digits after the decimal point.)</td>
</tr>
<tr>
<td></td>
<td>%e</td>
<td>Exponential notation, such as 3.141593e+00 (Use a precision operator to specify the number of digits after the decimal point.)</td>
</tr>
</tbody>
</table>
### Optional Operators

The optional identifier, flags, field width, precision, and operators further define the format of the output text.

- **Flags**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>'–'</code></td>
<td>Left-justify. Works with all specifiers.</td>
</tr>
<tr>
<td><strong>Example:</strong> <code>%–5.2f</code></td>
<td></td>
</tr>
<tr>
<td><strong>Example:</strong> <code>%–10s</code></td>
<td></td>
</tr>
<tr>
<td><code>'+'</code></td>
<td>Always print a sign character (+ or –) for any numeric value. Works with all specifiers except u, o, x, X, and s.</td>
</tr>
<tr>
<td><strong>Example:</strong> <code>%+5.2f</code></td>
<td></td>
</tr>
<tr>
<td><strong>Example:</strong> <code>%+10s</code></td>
<td></td>
</tr>
<tr>
<td><code>' '</code></td>
<td>Insert a space before the value. Works with all specifiers except u, o, x, X, and s.</td>
</tr>
<tr>
<td><strong>Example:</strong> <code>% 5.2f</code></td>
<td></td>
</tr>
</tbody>
</table>
Pad to field width with zeros before the value. Works with all specifiers except s.

**Example:** `%05.2f`

Modify selected numeric conversions:
- For `%o`, `%x`, or `%X`, print `0`, `0x`, or `0X` prefix.
- For `%f`, `%e`, or `%E`, print decimal point even when precision is 0.
- For `%g` or `%G`, do not remove trailing zeros or decimal point.

Works with all specifiers except `d`, `i`, `u`, and `s`.

**Example:** `%#5.0f`

### Field Width

Minimum number of characters to print.

The function pads to field width with spaces before the value unless otherwise specified by flags.

### Precision

<table>
<thead>
<tr>
<th>For <code>%f</code>, <code>%e</code>, or <code>%E</code></th>
<th>Number of digits to the right of the decimal point</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Example:</strong></td>
<td><code>%4f</code> prints pi as <code>'3.1416'</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><code>d</code>, <code>i</code>, <code>o</code>, <code>u</code>, <code>x</code>, <code>X</code></th>
<th>Minimum number of digits to be written. Outputs shorter than the specified precision are padded with leading zeros.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Example:</strong></td>
<td><code>%.4d</code> prints 5 as <code>'0005'</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>For <code>%g</code> or <code>%G</code></th>
<th>Number of significant digits</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Example:</strong></td>
<td><code>%4g</code> prints pi as <code>'3.142'</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><code>s</code></th>
<th>Maximum number of characters to be written to the output.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Example:</strong></td>
<td><code>%.2s</code> prints &quot;Hello!&quot; as &quot;He&quot;</td>
</tr>
</tbody>
</table>

**Note** If you specify a precision operator for floating-point values that exceeds the precision of the input numeric data type, the results might not match the input values to the precision you specified. The result depends on your computer hardware and operating system.
Text Before or After Formatting Operators

<table>
<thead>
<tr>
<th>Special Character</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single quotation mark</td>
<td>''</td>
</tr>
</tbody>
</table>
| Percent character | %
| Backslash | \\ |
| Alarm | \a |
| Backspace | \b |
| Form feed | \f |
| New line | \n |
| Carriage return | \r |
| Horizontal tab | \t |
| Vertical tab | \v |
| Character whose Unicode® numeric value can be represented by the hexadecimal number, N | \xN |
| **Example:** sprintf('\x5A') returns 'Z' |
| Character whose Unicode numeric value can be represented by the octal number, N | \N |
| **Example:** sprintf('\132') returns 'Z' |

**Format** can also include additional text before a percent sign, %, or after a conversion character. The text can be:

- Ordinary text to print.
- Special characters that you cannot enter as ordinary text. This table shows how to represent special characters in `formatSpec`.

**Length Specifiers**

The Format String block supports the `h` and `l` length subspecifiers. These specifiers can change according to the **Configuration Parameters > Hardware Implementation > Number of bits** settings.
<table>
<thead>
<tr>
<th>Length</th>
<th>d i</th>
<th>u o x X</th>
<th>f e E g G</th>
<th>s</th>
</tr>
</thead>
<tbody>
<tr>
<td>No length specifier</td>
<td>int</td>
<td>unsigned int</td>
<td>double (default), single</td>
<td>string</td>
</tr>
<tr>
<td>h</td>
<td>short</td>
<td>unsigned short</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>l</td>
<td>long</td>
<td>unsigned long</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>

**Note for Specifiers that Specify Integer Data Types (d, i, u, o, x, X)**

Target int, long, and short type sizes are controlled by settings in the Configuration Parameters > Hardware Implementation pane. For example, if the target int is 32 bits and the specifier is %u, then the expected input type will be uint32. However, the input port accepts any built-in integer type of that size or smaller with the %u specifier.

**Notes for Specifiers that Specify Floating Point Data Types (f, e, E, g, F)**

- Do not use l and h with these specifiers. Do not use the length subspecifier (for example, %f is allowed, but %hf and %lf are not allowed).
- Ports that correspond with these specifiers accept both single and double data types.

**Note for Specifiers that Specify the String Data Type (s)**

- The s specifier does not work with the l or h subspecifiers, and only accepts a string input data type.

**Extended Capabilities**

**C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.
See Also
ASCII to String | Scan String | String Compare | String Concatenate | String Constant | String Find | String Length | String to ASCII | String to Double | String to Enum | String to Single | Substring | To String | sprintf

Topics
“Display and Extract Coordinate Data”
“Simulink Strings”

 Introduced in R2018a
Configurable Subsystem

Represent any block selected from user-specified library of blocks

Library: Simulink / Ports & Subsystems

Description

Note Configurable Subsystem block will be removed in a future release. It is recommended to use Variant Subsystem instead of Configurable Subsystem. Variant Subsystems offer more capabilities than Configurable Subsystems with these advantages:

- You can mix Model blocks and Subsystem blocks as variant choices.
- You can specify variants that have different numbers of input and output ports.

For more information on converting Configurable Subsystem block to a Variant Subsystem block, see “Convert Configurable Subsystem to Variant Subsystem”.

For more information on Variant Subsystem, see Variant Subsystem, Variant Model.

The Configurable Subsystem block represents one of a set of blocks contained in a specified library of blocks. The context menu of the Configurable Subsystem block lets you choose which block the configurable subsystem represents.

Configurable Subsystem blocks simplify creation of models that represent families of designs. For example, suppose that you want to model an automobile that offers a choice of engines. To model such a design, you would first create a library of models of the engine types available with the car. You would then use a Configurable Subsystem block in your car model to represent the choice of engines. To model a particular variant of the basic car design, a user need only choose the engine type, using the configurable engine block's dialog box.

To create a configurable subsystem in a model, you must first create a library containing a master configurable subsystem and the blocks that it represents. You can then create
configurable instances of the master subsystem by dragging copies of the master subsystem from the library and dropping them into models.

You can add any type of block to a master configurable subsystem library. Simulink derives the port names for the configurable subsystem by making a unique list from the port names of all the choices. However, Simulink uses default port names for non-subsystem block choices.

You cannot break library links in a configurable subsystem because Simulink uses those links to reconfigure the subsystem when you choose a new configuration. Breaking links would be useful only if you do not intend to reconfigure the subsystem. In this case, you can replace the configurable subsystem with a nonconfigurable subsystem that implements the permanent configuration.

**Creating a Master Configurable Subsystem**

To create a master configurable subsystem:

1. Create a library of blocks representing the various configurations of the configurable subsystem.
2. Save the library.
3. Create an instance of the Configurable Subsystem block in the library.

   To do so, drag a copy of the Configurable Subsystem block from the Simulink Ports & Subsystems library into the library you created in the previous step.

4. Display the Configurable Subsystem block dialog box by double-clicking it. The dialog box displays a list of the other blocks in the library.

5. Under **List of block choices** in the dialog box, select the blocks that represent the various configurations of the configurable subsystems you are creating.

6. To apply the changes and close the dialog box, click the **OK** button.

7. Select **Block Choice** from the context menu of the Configurable Subsystem block.

   The context menu displays a submenu listing the blocks that the subsystem can represent.

8. Select the block that you want the subsystem to represent by default.

9. Save the library.
Note: If you add or remove blocks from a library, you must recreate any Configurable Subsystem blocks that use the library.

If you modify a library block that is the default block choice for a configurable subsystem, the change does not immediately propagate to the configurable subsystem. To propagate this change, do one of the following:

- Change the default block choice to another block in the subsystem, then change the default block choice back to the original block.
- Recreate the configurable subsystem block, including the selection of the updated block as the default block choice.

If a configurable subsystem in your model contains a broken link to a library block, editing the link and saving the model does not fix the broken link the next time you open the model. To fix a broken library link in your configurable subsystem, use one of the following approaches.

- Convert the configurable subsystem to a variant subsystem. Right-click the configurable subsystem, and select Subsystem and Model Reference > Convert Subsystem to > Variant Subsystem.
- Remove the library block from the master configurable subsystem library, add the library block back to the master configurable subsystem library, and then resave the master configurable subsystem library.

Creating an Instance of a Configurable Subsystem

To create an instance of a configurable subsystem in a model:

1. Open the library containing the master configurable subsystem.
2. Drag a copy of the master into the model.
3. Select Block Choice from the context menu of that Configurable Subsystem instance.
4. Select the block that you want the configurable subsystem to represent.

The instance of the configurable system displays the icon and parameter dialog box of the block that it represents.
Setting Instance Block Parameters

As with other blocks, you can use the parameter dialog box of a configurable subsystem instance to set its parameters interactively and the set_param command to set the parameters from the MATLAB command line or in a MATLAB file. If you use set_param, you must specify the full path name of the configurable subsystem's current block choice as the first argument of set_param, for example:

```matlab
curr_choice = get_param('mymod/myconfigsys', 'BlockChoice');
curr_choice = ['mymod/myconfigsys/' curr_choice];
set_param(curr_choice, 'MaskValues', ...);
```

Mapping I/O Ports

A configurable subsystem displays a set of input and output ports corresponding to input and output ports in the selected library. Simulink uses the following rules to map library ports to Configurable Subsystem block ports:

- Map each uniquely named input/output port in the library to a separate input/output port of the same name on the Configurable Subsystem block.
- Map all identically named input/output ports in the library to the same input/output ports on the Configurable Subsystem block.
- Terminate any input/output port not used by the currently selected library block with a Terminator/Ground block.

This mapping allows a user to change the library block represented by a Configurable Subsystem block without having to rewire connections to the Configurable Subsystem block.

For example, suppose that a library contains two blocks A and B and that block A has input ports labeled a, b, and c and an output port labeled d and that block B has input ports labeled a and b and an output port labeled e.
A Configurable Subsystem block based on this library would have three input ports labeled a, b, and c, respectively, and two output ports labeled d and e.

In this example, port a on the Configurable Subsystem block connects to port a of the selected library block no matter which block is selected. Port c on the Configurable Subsystem block functions only if library block A is selected. Otherwise, it simply terminates.

**Note** A Configurable Subsystem block does not provide ports that correspond to non-I/O ports, such as the trigger and enable ports on triggered and enabled subsystems. Thus,
you cannot use a Configurable Subsystem block directly to represent blocks that have such ports. You can do so indirectly, however, by wrapping such blocks in subsystem blocks that have input or output ports connected to the non-I/O ports.

**Convert to Variant Subsystem**

Right-click a configurable subsystem and select **Subsystems and Model Reference > Convert Subsystem To > Variant Subsystem**.

During conversion, Simulink performs the following operations:

- Replaces the Subsystem block with a Variant Subsystem block, preserving ports and connections.
- Adds the original subsystem as a variant choice in the Variant Subsystem block.
- Overrides the Variant Subsystem block to use the subsystem that was originally the active choice.
- Preserves links to libraries. For linked subsystems, Simulink adds the linked subsystem as a variant choice.

Simulink also preserves the subsystem block masks, and it copies the masks to the variant choice.

See Variant Subsystem for more information on variant choices.

**Ports**

**Input**

**Port_1 — Input signal**

scalar | vector | matrix | N-D array | bus

Input signal to the Configurable Subsystem. The block that the Configurable Subsystem represents determines the supported data types and dimensions of the input signal.

**Dependencies**

The number of input ports depends on the blocks in the library that the Configurable Subsystem represents. For more information, see “Mapping I/O Ports” on page 1-226.
Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | Boolean | fixed point | enumerated | bus

**Output**

**Port_1 — Output signal**
scalar | vector | matrix | N-D array | bus

Output signal from the Configurable Subsystem. The block that the Configurable Subsystem represents determines the output data types and dimensions.

**Dependencies**

The number of output ports depends on the blocks in the library that the Configurable Subsystem represents. For more information, see “Mapping I/O Ports” on page 1-226.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | Boolean | fixed point | enumerated | bus

**Parameters**

**List of block choices — Block members of the configurable subsystem**
no default

Select the blocks you want to include as members of the configurable subsystem. You can include user-defined subsystems as blocks.

**Programmatic Use**

**Block Parameter:** MemberBlocks

**Type:** cell array of character vectors

**Values:** cell array of block names as character vectors

**Default:** {''}

**Port names — Port names**
no default

Lists of input and output ports of member blocks. In the case of multiports, you can rearrange selected port positions by clicking the **Up** and **Down** buttons.
## Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>bus</th>
<th>double</th>
<th>enumerated</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
<th>string</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Direct Feedthrough</th>
<th>no</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multidimensional Signals</td>
<td>limited</td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>limited</td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
</tr>
</tbody>
</table>

a. Actual data type or capability support depends on block implementation.

## Extended Capabilities

### C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

Actual code generation support depends on block implementation.

### HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

### Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

Actual data type support depends on block implementation.
See Also
Subsystem, Atomic Subsystem, Nonvirtual Subsystem, CodeReuse Subsystem | Variant Subsystem

Topics
“Model Discretizer”

Introduced before R2006a
**Constant**

Generate constant value  
**Library:**  
Simulink / Commonly Used Blocks  
Simulink / Sources  
DSP System Toolbox / Sources  
HDL Coder / Commonly Used Blocks  
HDL Coder / Sources

**Description**

The Constant block generates a real or complex constant value signal. Use this block to provide a constant signal input. The block generates scalar, vector, or matrix output, depending on:

- The dimensionality of the **Constant value** parameter
- The setting of the **Interpret vector parameters as 1-D** parameter

The output of the block has the same dimensions and elements as the **Constant value** parameter. If you specify for this parameter a vector that you want the block to interpret as a vector, select the **Interpret vector parameters as 1-D** check box. Otherwise, if you specify a vector for the **Constant value** parameter, the block treats that vector as a matrix.

**Tip** To output a constant enumerated value, consider using the Enumerated Constant block instead. The Constant block provides block parameters that do not apply to enumerated types, such as **Output minimum** and **Output maximum**.

**Using Bus Objects as the Output Data Type**

The Constant block supports nonvirtual buses as the output data type. Using a bus object as the output data type can help simplify your model. If you use a bus object as the output data type, set the **Constant value** to θ or to a MATLAB structure that matches the bus object.
Using Structures for the Constant Value of a Bus

The structure you specify must contain a value for every element of the bus represented by the bus object. The block output is a nonvirtual bus signal.

You can use the `Simulink.Bus.createMATLABStruct` to create a full structure that corresponds to a bus.

You can use `Simulink.Bus.createObject` to create a bus object from a MATLAB structure.

If the signal elements in the output bus use numeric data types other than `double`, you can specify the structure fields by using typed expressions such as `uint16(37)` or untyped expressions such as `37`. To control the field data types, you can use the bus object as the data type of a `Simulink.Parameter` object. To decide whether to use typed or untyped expressions, see “Control Data Types of Initial Condition Structure Fields”.

Setting Configuration Parameters to Support Using a Bus Object Data Type

To enable the use of a bus object as an output data type, before you start a simulation, set `Configuration Parameters > Diagnostics > Data Validity > Advanced parameters > Underspecified initialization detection` to `Simplified`. For more information, see “Underspecified initialization detection”.

Ports

Output

Port_1 — Constant value

`scalar | vector | matrix | N-D array`

Constant value, specified as a real or complex valued scalar, vector, matrix, or N-D array. By default, the Constant block outputs a signal whose dimensions, data type, and complexity are the same as those of the Constant value parameter. However, you can specify the output to be any data type that Simulink supports, including fixed-point and enumerated data types.
**Note** If you specify a bus object as the data type for this block, do not set the maximum value for bus data on the block. Simulink ignores this setting. Instead, set the maximum values for bus elements of the bus object specified as the data type. For more information, see `Simulink.BusElement`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `Boolean` | `fixed point` | `enumerated` | `bus`

## Parameters

### Main

**Constant value — Constant output value**

1 (default) | scalar | vector | matrix | N-D array

Specify the constant value output of the block.

- You can enter any expression that MATLAB evaluates as a matrix, including the Boolean keywords `true` and `false`.
- If you set the **Output data type** to be a bus object, you can specify one of these options:
  - A full MATLAB structure corresponding to the bus object
  - 0 to indicate a structure corresponding to the ground value of the bus object
  
  For details, see “Using Bus Objects as the Output Data Type” on page 1-232.
- For nonbus data types, Simulink converts this parameter from its value data type to the specified output data type offline, using a rounding method of nearest and overflow action of saturate.

**Programmatic Use**

- **Block Parameter:** Value
  - **Type:** character vector
  - **Value:** scalar | vector | matrix | N-D array
  - **Default:** '1'

**Interpret vector parameters as 1-D — Treat vectors as 1-D**

on (default) | off
Select this check box to output a vector of length $N$ if the **Constant value** parameter evaluates to an $N$-element row or column vector.

- When you select this check box, the block outputs a vector of length $N$ if the **Constant value** parameter evaluates to an $N$-element row or column vector. For example, the block outputs a matrix of dimension 1-by-$N$ or $N$-by-1.
- When you clear this check box, the block does not output a vector of length $N$ if the **Constant value** parameter evaluates to an $N$-element row or column vector.

**Programmatic Use**

**Block Parameter:** VectorParams1D  
**Type:** character vector  
**Values:** 'on' | 'off'  
**Default:** 'on'

**Sample time — Sampling interval**

$\text{inf}$ (default) | scalar | vector

Specify the interval between times that the Constant block output can change during simulation (for example, due to tuning the **Constant value** parameter).

The default value of $\text{inf}$ indicates that the block output can never change. This setting speeds simulation and generated code by avoiding the need to recompute the block output.

See “Specify Sample Time” for more information.

**Programmatic Use**

**Block Parameter:** SampleTime  
**Type:** character vector  
**Values:** scalar | vector  
**Default:** 'inf'

**Signal Attributes**

**Output minimum — Minimum output value for range checking**

[] (default) | scalar

Specify the lower value of the output range that Simulink checks as a finite, real, double, scalar value.
**Note** If you specify a bus object as the data type for this block, do not set the minimum value for bus data on the block. Simulink ignores this setting. Instead, set the minimum values for bus elements of the bus object specified as the data type. For information on the Minimum parameter for a bus element, see `Simulink.BusElement`.

Simulink uses the minimum to perform:

- Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters”) for some blocks.
- Simulation range checking (see “Signal Ranges” and “Enable Simulation Range Checking”).
- Automatic scaling of fixed-point data types.
- Optimization of the code that you generate from the model. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. For more information, see “Optimize using the specified minimum and maximum values” (Simulink Coder).

**Note** Output minimum does not saturate or clip the actual output signal. Use the Saturation block instead.

**Programmatic Use**

**Block Parameter:** OutMin

**Type:** character vector

**Values:** scalar

**Default:** '[]'

**Output maximum — Maximum output value for range checking**

[] (default) | scalar

Specify the upper value of the output range that Simulink checks as a finite, real, double, scalar value.

**Note** If you specify a bus object as the data type for this block, do not set the maximum value for bus data on the block. Simulink ignores this setting. Instead, set the maximum values for bus elements of the bus object specified as the data type. For information on the Maximum parameter for a bus element, see `Simulink.BusElement`. 
Simulink uses the maximum value to perform:

- Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters”) for some blocks.
- Simulation range checking (see “Signal Ranges” and “Enable Simulation Range Checking”).
- Automatic scaling of fixed-point data types.
- Optimization of the code that you generate from the model. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. For more information, see “Optimize using the specified minimum and maximum values” (Simulink Coder).

**Note** Output maximum does not saturate or clip the actual output signal. Use the Saturation block instead.

**Programmatic Use**
**Block Parameter:** OutMax  
**Type:** character vector  
**Values:** scalar  
**Default:** '[]'

**Output data type — Output data type**
Inherit: Inherit from 'Constant value' (default) | Inherit: Inherit via back propagation | double | single | int8 | int16 | int32 | uint8 | uint16 | uint32 | boolean | fixdt(1,16) | fixdt(1,16,0) | fixdt(1,16,2^0,0) | Enum: <class name> | Bus: <object name> | <data type expression>

Specify the output data type. The type can be inherited, specified directly, or expressed as a data type object such as Simulink.NumericType.

Click the **Show data type assistant** button to display the **Data Type Assistant**, which helps you set the data type attributes. For more information, see “Specify Data Types Using Data Type Assistant”.

**Programmatic Use**
**Block Parameter:** OutDataTypeStr  
**Type:** character vector
Values: 'Inherit: Inherit from 'Constant value'' | 'Inherit: Inherit via back propagation' | 'double' | 'single' | 'int8' | 'uint8' | 'int16' | 'uint16' | 'int32' | 'uint32' | 'boolean' | 'fixdt(1,16)' | 'fixdt(1,16,0)' | 'fixdt(1,16,2^0,0)' | 'Enum: <class name>' | 'Bus: <object name>'

Default: 'Inherit: Inherit from 'Constant value''

Lock output data type setting against changes by the fixed-point tools — Prevent fixed-point tools from overriding Output data type

off (default) | on

Select this parameter to prevent the fixed-point tools from overriding the Output data type you specify on the block. For more information, see “Use Lock Output Data Type Setting” (Fixed-Point Designer).

Programmatic Use
Block Parameter: LockScale
Type: character vector
Values: 'off' | 'on'
Default: 'off'

Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>bus</th>
<th>double</th>
<th>enumerated</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

Tunable Parameters
You can use a tunable parameter in a Constant block intended for HDL code generation. For details, see “Generate DUT Ports for Tunable Parameters” (HDL Coder).

HDL Architecture

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>default</td>
<td>None</td>
<td>This implementation emits the value of the Constant block.</td>
</tr>
<tr>
<td>Logic Value</td>
<td>None</td>
<td>By default, this implementation emits the character 'Z' for each bit in the signal. For example, for a 4-bit signal, the implementation would emit 'ZZZZ'.</td>
</tr>
<tr>
<td></td>
<td>{'Value', 'Z'}</td>
<td>If the signal is in a high-impedance state, use this parameter value. This implementation emits the character 'Z' for each bit in the signal. For example, for a 4-bit signal, the implementation would emit 'ZZZZ'.</td>
</tr>
<tr>
<td></td>
<td>{'Value', 'X'}</td>
<td>If the signal is in an unknown state, use this parameter value. This implementation emits the character 'X' for each bit in the signal. For example, for a 4-bit signal, the implementation would emit 'XXXX'.</td>
</tr>
</tbody>
</table>
HDL Block Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConstrainedOutput Pipeline</td>
<td>Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. For more details, see “ConstrainedOutputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td>InputPipeline</td>
<td>Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “InputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td>OutputPipeline</td>
<td>Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “OutputPipeline” (HDL Coder).</td>
</tr>
</tbody>
</table>

Complex Data Support

This block supports code generation for complex signals.

Restrictions

- The Logic Value implementation does not support the double data type. If you specify this implementation for a constant value of type double, a code generation error occurs.
- For Sample time, enter -1. Delay balancing does not support an inf sample time.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

Design and simulate fixed-point systems using Fixed-Point Designer™.

See Also

Enumerated Constant | Simulink.BusElement | Simulink.Parameter
Topics
“Set Block Parameter Values”
“Specify Bus Properties with Bus Objects”
“Specify Initial Conditions for Bus Signals”
“Group Constant Signals into an Array of Buses”

Introduced before R2006a
Coulomb and Viscous Friction

Model discontinuity at zero, with linear gain elsewhere

Library:  
Simulink / Discontinuities  
HDL Coder / Discontinuities

Description

The Coulomb and Viscous Friction block models Coulomb (static) and viscous (dynamic) friction. The block models a discontinuity at zero and a linear gain otherwise.

The block output matches the MATLAB result for:

\[ y = \text{sign}(x) \times (\text{Gain} \times \text{abs}(x) + \text{Offset}) \]

where \( y \) is the output, \( x \) is the input, \( \text{Gain} \) is the signal gain for nonzero input values, and \( \text{Offset} \) is the Coulomb friction.

The block accepts one input and generates one output. The input can be a scalar, vector, or matrix with real and complex elements.

- For a scalar input, \( \text{Gain} \) and \( \text{Offset} \) can have dimensions that differ from the input. The output is a scalar, vector, or matrix depending on the dimensions of \( \text{Gain} \) and \( \text{Offset} \).
- For a vector or matrix input, \( \text{Gain} \) and \( \text{Offset} \) must be scalar or have the same dimensions as the input. The output is a vector or matrix of the same dimensions as the input.
Ports

Input
Port_1 — Input signal
scalar | vector | matrix

The input signal to the model of Coulomb and viscous friction.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

Output
Port_1 — Output signal
scalar | vector | matrix

The output signal calculated by applying the friction models to the input.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

Parameters

To edit the parameters for the Coulomb and Viscous Friction block, double-click the block icon.

Coulomb friction value — Static friction offset
[1320] (default) | real values

Specify the offset that applies to all input values.

Programmatic Use
Block Parameter: offset
Type: character vector
Value: real values
Default: ' [1 3 2 0]' 

Coefficient of viscous friction — Dynamic friction coefficient
1 (default) | real values
Specify the signal gain for nonzero input values.

**Programmatic Use**

**Block Parameter:** gain  
**Type:** character vector  
**Value:** real values  
**Default:** '1'

### Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>double</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Extended Capabilities

**C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

**HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.
**HDL Architecture**

This block has a single, default HDL architecture.

**HDL Block Properties**

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConstrainedOutput Pipeline</td>
<td>Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. For more details, see “ConstrainedOutputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td>InputPipeline</td>
<td>Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “InputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td>OutputPipeline</td>
<td>Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “OutputPipeline” (HDL Coder).</td>
</tr>
</tbody>
</table>

**Restrictions**

HDL code generation does not support complex input.

**PLC Code Generation**

Generate Structured Text code using Simulink® PLC Coder™.

**Fixed-Point Conversion**

Design and simulate fixed-point systems using Fixed-Point Designer™.

**See Also**

Backlash | Dead Zone

**Introduced before R2006a**
Counter Free-Running

Count up and overflow back to zero after reaching maximum value for specified number of bits

**Library:**
- Simulink / Sources
- HDL Coder / Sources

**Description**

The Counter Free-Running block counts up until reaching the maximum value, $2^{N\text{bits}} - 1$, where $N\text{bits}$ is the number of bits. Then the counter overflows to zero and begins counting up again.

After overflow, the counter always initializes to zero. However, if you select the global doubles override, the Counter Free-Running block does not wrap back to zero.

**Note** This block does not report wrap on overflow warnings during simulation. To report these warnings, see the Simulink.restoreDiagnostic reference page. The block does report errors due to wrap on overflow.

**Ports**

**Output**

**Port_1 — Count value**

scalar

Count value, specified as an unsigned integer of 8 bits, 16 bits, or 32 bits.

Data Types: `uint8` | `uint16` | `uint32`
Parameters

**Number of Bits — Number of bits**  
16 (default) | scalar

Specify the number of bits as a finite, real-valued. When you specify:

- A positive integer, for example 8, the block counts up to $2^8 - 1$, which is 255.
- An unsigned integer expression, for example uint8(8), the block counts up to uint8($2^{\text{uint8}(8)} - 1$), which is 254.

**Programmatic Use**  
**Block Parameter:** NumBits  
**Type:** character vector  
**Values:** scalar  
**Default:** '16'

**Sample time — Interval between samples**  
-1 (default) | scalar | vector

Specify the time interval between samples as a scalar (sampling period), or a two-element vector ([sampling period, initial offset]). To inherit the sample time, set this parameter to -1. For more information, see “Specify Sample Time”.

**Programmatic Use**  
**Block Parameter:** tsamp  
**Type:** character vector  
**Values:** scalar | vector  
**Default:** ' -1'

Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>fixed point</th>
<th>integer</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Direct Feedthrough</strong></td>
<td>no</td>
<td></td>
</tr>
<tr>
<td><strong>Multidimensional Signals</strong></td>
<td>no</td>
<td></td>
</tr>
</tbody>
</table>
Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

Not recommended for production code. Relates to resource limits and restrictions on speed and memory often found in embedded systems. Generated code can contain dynamic allocation and freeing of memory, recursion, additional memory overhead, and widely-varying execution times. While the code is functionally valid and generally acceptable in resource-rich environments, smaller embedded targets often cannot support such code. Usually, blocks evolve toward being suitable for production code. Thus, blocks suitable for production code remain suitable.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture
This block has a single, default HDL architecture.

HDL Block Properties

| ConstrainedOutput Pipeline | Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. For more details, see “ConstrainedOutputPipeline” (HDL Coder). |
### InputPipeline
Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “InputPipeline” (HDL Coder).

### OutputPipeline
Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “OutputPipeline” (HDL Coder).

---

**PLC Code Generation**
Generate Structured Text code using Simulink® PLC Coder™.

**Fixed-Point Conversion**
Design and simulate fixed-point systems using Fixed-Point Designer™.

**See Also**
Counter Limited | MATLAB Function

**Introduced before R2006a**
Counter Limited

Count up and wrap back to zero after outputting specified upper limit

Library: Simulink / Sources
        HDL Coder / Sources

Description

The Counter Limited block counts up until the specified upper limit is reached. Then the counter wraps back to zero, and restarts counting up. The counter always initializes to zero.

Note

This block does not report wrap on overflow warnings during simulation. To report these warnings, see the Simulink.restoreDiagnostic reference page. The block does report errors due to wrap on overflow.

Ports

Output

Port 1 — Count value
scalar

Count value, specified as an unsigned integer of 8 bits, 16 bits, or 32 bits. The block uses the smallest number of bits required to represent the upper limit.

Data Types: uint8 | uint16 | uint32

Parameters

Upper limit — Upper limit
7 (default) | scalar

Specify the upper limit for the block to count to as a finite, real-valued scalar.
**Programmatic Use**

**Block Parameter:** uplimit
**Type:** character vector
**Values:** scalar
**Default:** '7'

**Sample time — Interval between samples**
-1 (default) | scalar | vector

Specify the time interval between samples as a scalar (sampling period), or a two-element vector (sampling period, initial offset). To inherit the sample time, set this parameter to -1. For more information, see “Specify Sample Time”.

**Programmatic Use**

**Block Parameter:** tsamp
**Type:** character vector
**Values:** scalar | vector
**Default:** ' -1'

**Block Characteristics**

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>fixed point</th>
<th>integer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>no</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

- The code generator does not explicitly group primitive blocks that constitute a nonatomic masked subsystem block in the generated code. This flexibility allows for more efficient code generation. In certain cases, you can achieve grouping by configuring the masked subsystem block to execute as an atomic unit by selecting the Treat as atomic unit option.
- Not recommended for production code. Relates to resource limits and restrictions on speed and memory often found in embedded systems. Generated code can contain dynamic allocation and freeing of memory, recursion, additional memory overhead, and widely-varying execution times. While the code is functionally valid and generally acceptable in resource-rich environments, smaller embedded targets often cannot support such code. Usually, blocks evolve toward being suitable for production code. Thus, blocks suitable for production code remain suitable.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture
This block has a single, default HDL architecture.

HDL Block Properties

| ConstrainedOutput Pipeline | Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. For more details, see “ConstrainedOutputPipeline” (HDL Coder). |
InputPipeline
Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “InputPipeline” (HDL Coder).

OutputPipeline
Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “OutputPipeline” (HDL Coder).

PLC Code Generation
Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

See Also
Counter Free-Running

Introduced before R2006a
Custom Gauge

Display an input value on a customized gauge

**Library:** Simulink / Dashboard

**Description**

The Custom Gauge block displays the value of the connected signal on a gauge face that you can customize to look like a gauge in a real system. For example, you could create an engine rpm gauge or fuel indicator for an automotive model.

The Custom Gauge block provides an indication of the instantaneous value of the connected signal throughout simulation. You can modify the range and tick values on the Custom Gauge block to fit your data. Use the Custom Gauge block with other Dashboard blocks to build an interactive dashboard of controls and indicators for your model.
Create A Custom Gauge

When you add a Custom Gauge block to your model, the block is preconfigured with a default design. You can use the preconfigured Custom Gauge block like you would other Dashboard blocks, or you can enter Custom Gauge edit mode to customize the appearance. The **Edit Custom Gauge** button appears above the Custom Gauge block when you pause on it.

In Custom Gauge edit mode, the customization menu appears above the block. Separate images specify the appearance of the needle and the background. You can also customize the size of the dial arc and the color and transparency of the arc and tick marks. To build from the default design, you can individually select the needle image, background image, and arc to reposition and resize each feature. If you want to start from a blank canvas, you can remove the arc with the **Clear Arc** button. You can remove the needle and background images by clicking the X that appears when you pause on the needle or background image buttons.
The blank canvas for the Custom Gauge block shows a circle with its center marked to help guide alignment as you build your customized gauge. You can drag the center marker to align the center of your arc with the center of the arc in your background image. To add an image for the needle or the gauge background, click the button for the type of image you want to add. Then, select the image you want to use. When you add a gauge background image, the Custom Gauge block **Lock Aspect Ratio** parameter is selected. You can resize the block without distorting the image. If you want to change the aspect ratio of the block after adding a gauge background image, clear the **Lock Aspect Ratio** parameter.
To draw the arc for your gauge, click and drag along the guide circle. As you draw the arc, crosshairs appear to assist you with aligning the start and end points of the arc. After you release the mouse, you can continue to adjust the arc position and size. The arc options on the toolbar control the appearance of the arc, tick marks, and the value arc that indicates the value of the connected signal or parameter during simulation. The slider adjusts the transparency of the selected feature, and the color wheel allows you to specify the color. You can configure the arc as fully transparent when your background image includes the representation of the gauge face you want to use.
When you finish editing the appearance of your Custom Gauge block, click the **Exit** button in the upper-right of the model canvas or press **Esc** to exit Custom Gauge edit mode.

### Connecting Dashboard Blocks

Dashboard blocks do not use ports to connect to signals. To connect Dashboard blocks to signals in your model, use connect mode. Connect mode facilitates the process of connecting Dashboard blocks in your model, especially when you want to connect multiple blocks at once. If you only want to connect a single Dashboard block, you can also use the **Connection** table in the block dialog box to make the connection.

**Tip** To facilitate understanding and debugging your model, you can modify Dashboard block connections in your model during normal and accelerator mode simulations.

To enter connect mode, click the **Connect** button that appears above your unconnected Dashboard block when you pause on it.
In connect mode, when you select one or more signals or blocks, a list of parameters and signals available for connection appears. Select a signal or parameter from the list to connect the selected Dashboard block. To connect another Dashboard block, pause on the block and click the **Connect** button above it. Then, make a selection of signals and blocks in your model, and choose a signal or parameter to connect.
When you finish connecting the Dashboard blocks in your model, click the **Exit** button in the upper-right of the canvas to exit connect mode.

**Tip** You can hide the message shown on unconnected blocks using the `set_param` function and the `ShowInitialText` block parameter.

**Limitations**

- Dashboard blocks can only connect to real scalar signals.
- You cannot use the **Connection** table to connect a Dashboard block to a block that is commented out. When you connect a Dashboard block to a commented block using connect mode, the Dashboard block does not display the connected value until you uncomment the block.
- Dashboard blocks cannot connect to signals inside referenced models.
- When you simulate a model hierarchy, Dashboard blocks inside referenced models do not update.
- Dashboard blocks do not support rapid accelerator simulation.
Some signals are not available during accelerator mode simulations. To view such signals on a Dashboard block, mark the signal for logging.

**Parameters**

**Connection — Select a signal to connect and display**

Select the signal to connect using the **Connection** table. Populate the **Connection** table by selecting signals of interest in your model. Select the radio button next to the signal you want to display. Click **Apply** to connect the signal. To facilitate understanding and debugging your model, you can connect Dashboard blocks to signals in your model during simulation.

**Programmatic Use**

**Block Parameter:** Binding  
**Type:** `Simulink.HMI.SignalSpecification`  
**Default:** `[]`

**Minimum — Minimum tick mark value**

0 (default) | scalar

Finite, real, double, scalar value specifying the minimum tick mark value for the arc. The minimum must be less than the value entered for the maximum.

**Programmatic Use**

To programmatically set the Minimum parameter, use a 1-by-3 vector containing values for the Minimum, Tick Interval, and Maximum parameters, in that order. To use the auto value for the Tick Interval, leave the Tick Interval position in the vector empty or specify -1.

**Block Parameter:** Limits  
**Type:** 1x3 vector  
**Default:** `[0 -1 100]`

**Maximum — Maximum tick mark value**

100 (default) | scalar

Finite, real, double, scalar value specifying the maximum tick mark value for the arc. The maximum must be greater than the value entered for the minimum.
**Programmatic Use**

To programmatically set the **Maximum** parameter, use a 1-by-3 vector containing values for the **Minimum**, **Tick Interval**, and **Maximum** parameters, in that order. To use the **auto** value for the **Tick Interval**, leave the **Tick Interval** position in the vector empty or specify -1.

**Block Parameter:** Limits  
**Type:** 1x3 vector  
**Default:** [0 -1 100]

**Tick Interval — Interval between major tick marks**  
*auto* (default) | scalar

Finite, real, positive, integer, scalar value specifying the interval of major tick marks on the arc. When set to **auto**, the block automatically adjusts the tick interval based on the minimum and maximum values.

**Programmatic Use**

To programmatically set the **Tick Interval** parameter, use a 1-by-3 vector containing values for the **Minimum**, **Tick Interval**, and **Maximum** parameters, in that order. To use the **auto** value for the **Tick Interval**, leave the **Tick Interval** position in the vector empty or specify -1.

**Block Parameter:** Limits  
**Type:** 1x3 vector  
**Default:** [0 -1 100]

**Scale Colors — Color indications on gauge arc**  
*colors for arc ranges*

Color specifications for ranges on the arc. Press the + button to add a color. For each color added, specify the minimum and maximum values of the range where you want to display that color.

**Programmatic Use**

To programmatically specify the **Scale Colors** parameter, use an array of structures with fields:

- **Min** — Minimum value for the color range on the scale.
- **Max** — Maximum value for the color range on the scale.
- **Color** — 1-by-3 vector of **double** values between 0 and 1 to specify the color for the range of the scale in the form 

\[
[r \ g \ b]
\]
Include a structure in the array for each scale range for which you want to specify a color.

```matlab
range1.Min = 0;
range1.Max = 10;
range1.Color = [0 0 1];
range2.Min = 10;
range2.Max = 15;
range2.Color = [0 1 0];
scaleRanges = [range1 range2];
```

**Block Parameter:** ScaleColors  
**Type:** structure array  
**Default:** 0x1 struct array

**Label — Block label position**  
**Top** (default) | **Bottom** | **Hide**

Position of the block label. When the block is connected to an element in the model, the label is the name of the connected element.

**Programmatic Use**  
**Block Parameter:** LabelPosition  
**Type:** character vector  
**Values:** 'Top' | 'Bottom' | 'Hide'  
**Default:** 'Top'

**Lock Aspect Ratio — Maintain background image aspect ratio**  
**on** (default) | **off**

Maintain the background image aspect ratio when resizing the block. By default, when you specify a background image for the block, the Lock Aspect Ratio check box is selected.

**Block Characteristics**

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>double</th>
<th>enumerated</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Extended Capabilities

#### C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

Dashboard blocks are ignored for code generation.

### See Also

Gauge | Half Gauge | Linear Gauge | Quarter Gauge

### Topics

“Tune and Visualize Your Model with Dashboard Blocks”
“Decide How to Visualize Simulation Data”

Introduced in R2018b
Dashboard Scope

Trace signals during simulation

Library: Simulink / Dashboard

Description

The Dashboard Scope block shows connected signals during simulation on a scope display. You can use the Dashboard Scope block with other Dashboard blocks to build an interactive dashboard of controls and indicators for your model. The Dashboard Scope block provides a complete picture of a signal's behavior over the course of the simulation. Use the Dashboard Scope block to display signals of any data type that Simulink supports, including enumerated data types. The Dashboard Scope block can display up to eight signals from a matrix or bus.

Connecting Dashboard Blocks

Dashboard blocks do not use ports to connect to signals. To connect Dashboard blocks to signals in your model, use connect mode. Connect mode facilitates the process of connecting Dashboard blocks in your model, especially when you want to connect multiple blocks at once. If you only want to connect a single Dashboard block, you can also use the Connection table in the block dialog box to make the connection.

Tip To facilitate understanding and debugging your model, you can modify Dashboard block connections in your model during normal and accelerator mode simulations.

To enter connect mode, click the Connect button that appears above your unconnected Dashboard block when you pause on it.
In connect mode, when you select one or more signals or blocks, a list of parameters and signals available for connection appears. Select a signal or parameter from the list to connect the selected Dashboard block. To connect another Dashboard block, pause on the block and click the Connect button above it. Then, make a selection of signals and blocks in your model, and choose a signal or parameter to connect.
When you finish connecting the Dashboard blocks in your model, click the **Exit** button in the upper-right of the canvas to exit connect mode.

**Tip** You can hide the message shown on unconnected blocks using the `set_param` function and the `ShowInitialText` block parameter.

**Data Cursors**

You can add data cursors to a Dashboard Scope block to inspect the displayed signals. The data cursors show when the Dashboard Scope block is selected. With the Dashboard Scope block selected, you can move cursors along the displayed signals to see the data values corresponding to each time sample. When you display two cursors, a box between the cursors along the time axis displays the time difference between the two cursors.
To add data cursors, right-click the Dashboard Scope block. Under the **Data Cursors** menu, select the number of cursors you want to add.

**Zoom and Pan**

You can also zoom and pan to inspect your signals. To change zoom and pan modes, right-click the Dashboard Scope block, and select the zoom or pan mode you want.

**Signal Appearance**

You can modify the color and line style for signals connected to a Dashboard Scope block from the **Connection** table in the block dialog box. To modify the appearance of a connected signal:

1. Click the signal appearance preview for the signal in the **Style** column of the **Connection** table.
2 Choose a signal color from the palette of standard colors or specify a custom color using RGB values between 0 and 255.

3 Select the line style from the solid, dotted, dashed, and dot-dashed options under the **Style** section in the signal appearance dialog.

Signal color and line style remain consistent for a signal throughout a model. You can configure the line style and color for a signal through the Instrumentation Properties for a logged signal and through the Dashboard Scope block dialog of a Dashboard Scope block connected to the signal. The **Override style and color** setting is selected by default and controls whether signal appearance options chosen through that Dashboard Scope block dialog box override signal appearance options configured elsewhere in the model. When you clear the **Override style and color** setting, changes made using the dialog box for that Dashboard Scope block are not applied.

When you connect signals to a Dashboard Scope block using the block dialog box, the connection table shows the default signal color and line style used for a signal you connect. As you check signals to connect, the **Connection** table updates the default style and color to be used for the next signal you connect from the remaining signals in the selection.
Complex Signals

The Dashboard Scope block displays complex signals according to their Complex Format. You can configure the Complex Format for a signal using the Instrumentation Properties dialog box, accessible when you right-click the logging badge for the signal and select Properties.

A signal can have a Complex Format that is:

- Real-Imaginary
- Magnitude-Phase
- Magnitude
- Phase

When you set the Complex Format for a signal to Real-Imaginary or Magnitude-Phase, the Dashboard Scope block displays both components of the signal. The real or magnitude component displays in the color indicated in the Connection table. The imaginary or phase component displays in a different shade of the color indicated in the Connection table.
Limitations

- You cannot use the Connection table to connect a Dashboard block to a block that is commented out. When you connect a Dashboard block to a commented block using connect mode, the Dashboard block does not display the connected value until the you uncomment the block.
- Dashboard blocks cannot connect to signals inside referenced models.
- When you simulate a model hierarchy, Dashboard blocks inside referenced models do not update.
- Dashboard blocks do not support rapid accelerator simulation.
- Some signals are not available during accelerator mode simulations. To view such signals on a Dashboard block, mark the signal for logging.
- You cannot access the context menu for the Dashboard Scope block when the block is in a panel. As a result, you cannot zoom, pan, fit to view, or add cursors.

Parameters

Connection — Select signals to connect and display
signal connection options

Select one or more signals to connect using the Connection table. Populate the Connection table by selecting signals of interest in your model. Select the check box next to the signal you want to display. Click Apply to connect the signal. To facilitate understanding and debugging your model, you can connect signals to a Dashboard Scope block during simulation.

You can also configure the line style and color for signals connected to a Dashboard Scope block using the Connection table. For more information, see “Signal Appearance” on page 1-268.

Programmatic Use

To programmatically specify one or more signals to connect to the Dashboard Scope block, use a cell array of Simulink.HMI.SignalSpecification objects.

Block Parameter: Binding
Type: cell array of Simulink.HMI.SignalSpecification objects
Default: []
To programmatically configure the line style and color for connected signals, use the Colors parameter. Specify the value for the Colors parameter as an array of structures, where each structure specifies the line style and color for one signal. Settings specified using the Colors parameter apply to the signal at the same index in the value for the Binding parameter.

Specify the line style and color for each signal as a structure with the fields:

- **Color** — 1-by-3 \([r \ g \ b]\) vector with values between 0 and 1 specifying the signal color.
- **LineStyle** — Character vector specifying one of these line style options:
  - Solid: `'-'`
  - Dashed: `'—'`
  - Dotted: `':'`
  - Dash-dotted: `'-.'

```matlab
sig1Colors.Color = [0 0 1];
sig1Colors.LineStyle = '--';
sig2Colors.Color = [1 0 1];
sig2Colors.LineStyle = '-';
sigColors = [sig1Colors sig2Colors];
```

**Block Parameter:** Colors  
**Type:** array of structures

**Time Span — Set horizontal axis span**  
**auto** (default) | scalar

A finite, real, double, scalar value that sets the time span of the scope display.

When **Time Span** is set to **auto**, the block sets its time span to the model's simulation stop time.

**Programmatic Use**  
**Block Parameter:** TimeSpan  
**Type:** string or character array  
**Values:** 'auto' or numeric value  
**Default:** 'auto'

**Min — Set y-axis minimum**  
-3 (default) | scalar
A finite, real, double, scalar value that sets the minimum of the vertical axis on the scope display.

Dependencies

The **Min** value must be less than the **Max** value.

Programmatic Use

To specify the **Min** parameter for the Dashboard Scope block programmatically, use a 1-by-2 vector containing the minimum y-axis value and maximum y-axis value, in that order.

**Block Parameter:** YLimits
**Type:** 1x2 vector
**Default:** [-3 3]

**Max — Set y-axis maximum**

A finite, real, double, scalar value that sets the maximum of the vertical axis on the scope display.

Dependencies

The **Max** value must be greater than the **Min** value.

Programmatic Use

To specify the **Max** parameter for the Dashboard Scope block programmatically, use a 1-by-2 vector containing the minimum y-axis value and maximum y-axis value, in that order.

**Block Parameter:** YLimits
**Type:** 1x2 vector
**Default:** [-3 3]

Legend — Set position of legend

**Top** (default) | **Right** | **Hide**

Options from the drop-down menu specify the position of the legend in the scope display. The legend shows the color chosen for each connected signal next to the signal's name.

Programmatic Use

**Block Parameter:** LegendPosition
**Type:** string or character vector
**Values:** 'Top' | 'Right' | 'Hide'
**Default:** 'Top'
**Scale axes limits at stop — Autoscale axes limits when simulation finishes**

on (default) | off

When on, performs a fit-to-view operation on the data displayed in the scope when the simulation stops.

**Programmatic Use**

Block Parameter: ScaleAtStop

**Type:** string or character vector

**Values:** 'on' | 'off'

**Default:** 'on'

**Show "Double-click to connect" message — Show or hide message**

on (default) | off

When on, shows instructional text if the block is not connected. When the block is not connected, you can specify this parameter as off to hide the text.

**Programmatic Use**

Block Parameter: ShowInitialText

**Type:** string or character vector

**Values:** 'on' | 'off'

**Default:** 'on'

## Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>bus</th>
<th>double</th>
<th>enumerated</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
<th>string</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.
Dashboard blocks are ignored for code generation.

See Also

Topics
“Tune and Visualize Your Model with Dashboard Blocks”
“Decide How to Visualize Simulation Data”

Introduced in R2015a
Data Store Memory

Define data store

Library: Simulink / Signal Routing

Description

The Data Store Memory block defines and initializes a named shared data store, which is a memory region usable by Data Store Read and Data Store Write blocks that specify the same data store name.

The location of the Data Store Memory block that defines a data store determines which Data Store Read and Data Store Write blocks can access the data store:

- If the Data Store Memory block is in the top-level system, Data Store Read and Data Store Write blocks anywhere in the model can access the data store.
- If the Data Store Memory block is in a subsystem, Data Store Read and Data Store Write blocks in the same subsystem or in any subsystem below it in the model hierarchy can access the data store.

Data Store Read or Data Store Write blocks cannot access a Data Store Memory block that is either in a model that contains a Model block or in a referenced model.

Do not include a Data Store Memory block in a For Each subsystem.

Obtaining correct results from data stores requires ensuring that data store reads and writes occur in the expected order. For details, see:

- “Order Data Store Access”
- “Data Store Diagnostics”
- “Log Data Stores”

You can use Simulink.Signal objects in addition to, or instead of, Data Store Memory blocks to define data stores. A data store defined in the base workspace with a signal
object is a global data store. Global data stores are accessible to every model, including all referenced models. See “Data Stores” for more information.

**Parameters**

**Main**

**Data store name — Name for the data store**
A (default) | character vector | string

Specify a name for the data store you are defining with this block. Data Store Read and Data Store Write blocks with the same name can read from, and write to, the data store initialized by this block. The name can represent a Data Store Memory block or a sign object defined to be a data store.

**Programmatic Use**

**Block Parameter:** DataStoreName  
**Type:** character vector  
**Values:** 'A' | ...  
**Default:** 'A'

**Rename All — Rename this data store throughout the model**

button

Rename this data store everywhere the Data Store Read and Data Store Write blocks use it in a model.

**Limitations**

You cannot use **Rename All** to rename a data store if you:

- Use a Simulink.Signal object in a workspace to control the code generated for the data store
- Use a Simulink.Signal object instead of a Data Store Memory block to define the data store

You must instead rename the corresponding Simulink.Signal object from Model Explorer. For an example, see “Rename Data Store Defined by Signal Object”.
Corresponding Data Store Read/Write blocks — Path to connected Data Store Read/Write blocks

block path

List all the Data Store Read and Data Store Write blocks that have the same data store name as the current block, and that are in the current system or in any subsystem below it in the model hierarchy. Clicking a block path displays and highlights that block in your model.

Signal Attributes

Initial value — Initial value of data store

Specify the initial value or values of the data store. The Minimum parameter specifies the minimum value for this parameter, and the Maximum parameter specifies the maximum value.

If you specify a nonscalar value and set Dimensions to -1 (the default), the data store has the same dimensions as the array. Data that you write to the data store (by using Data Store Write blocks) must have these dimensions.

If you set the Dimensions parameter to a value other than -1, the initial value dimensions must match the dimensions that you specify, unless the initial value is a scalar or a MATLAB structure. If you specify a scalar, each element of the data store uses the scalar as the initial value. Use this technique to apply the same initial value (the scalar that you specify) to each element without manually matching the dimensions of the initial value with the dimensions of the data store.

To use this block to initialize a nonvirtual bus signal, specify the initial value as a MATLAB structure and set the model configuration parameter “Underspecified initialization detection” to Simplified. For more information about initializing nonvirtual bus signals using structures, see “Specify Initial Conditions for Bus Signals”.

Programmatic Use

Block Parameter: InitialValue

Type: character vector

Values: scalar | vector | matrix | N-D array

Default: '0'

Minimum — Minimum output value for range checking

[] (default) | scalar
Specify the minimum value that the block should output. The default value is [ ] (unspecified). This number must be a finite real double scalar value.

**Note** If you specify a bus object as the data type for this block, do not set the minimum value for bus data on the block. Simulink ignores this setting. Instead, set the minimum values for bus elements of the bus object specified as the data type. For information on the Minimum property of a bus element, see `Simulink.BusElement`.

Simulink uses the minimum value to perform:

- Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters”).
- Simulation range checking (see “Signal Ranges”).
- Automatic scaling of fixed-point data types.
- Optimization of the code that you generate from the model. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. For more information, see “Optimize using the specified minimum and maximum values” (Simulink Coder).

**Programmatic Use**

**Block Parameter:** `OutMin`

**Type:** character vector

**Values:** scalar

**Default:** ' [ ] '

**Maximum — Maximum output value for range checking**

[ ] (default) | scalar

Specify the maximum value that the block should output. The default value is [ ] (unspecified). This number must be a finite real double scalar value.

**Note** If you specify a bus object as the data type for this block, do not set the maximum value for bus data on the block. Simulink ignores this setting. Instead, set the maximum values for bus elements of the bus object specified as the data type. For information on the Maximum property of a bus element, see `Simulink.BusElement`.

Simulink uses the maximum value to perform:
- Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters”).
- Simulation range checking (see “Signal Ranges”).
- Automatic scaling of fixed-point data types.
- Optimization of the code that you generate from the model. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. For more information, see “Optimize using the specified minimum and maximum values” (Simulink Coder).

**Programmatic Use**

**Block Parameter:** OutMax  
**Type:** character vector  
**Values:** scalar  
**Default:** '[]'

**Data type — Output data type**

Inherit: auto (default) | double | single | int8 | uint8 | int16 | uint16 | int32 | uint32 | int64 | uint64 | boolean | fixdt(1,16,0) | fixdt(1,16,2^0,0) | Enum: <class name>

Specify the output data type. You can set it to:

- A rule that inherits a data type (for example, Inherit: auto).
- The name of a built-in data type (for example, single).
- The name of a data type object (for example, a Simulink.NumericType object).
- An expression that evaluates to a data type (for example, fixdt(1,16,0)). Do not specify a bus object as the data type in an expression; use Bus: <object name> to specify a bus data type.

Click the **Show data type assistant** button to display the **Data Type Assistant**, which helps you set the data type attributes. For more information, see “Specify Data Types Using Data Type Assistant”.

**Programmatic Use**

**Block Parameter:** OutDataTypeStr  
**Type:** character vector  
**Values:** 'Inherit: auto' | 'double' | 'single' | 'int8' | 'uint8' | 'int16' | 'uint16' | 'int32' | 'uint32' | 'int64' | 'uint64' |
'boolean' | 'fixdt(1,16,0)' | 'fixdt(1,16,2^0,0)' | 'Enum: <class name>'

**Default:** 'Inherit: auto'

**Lock output data type setting against changes by the fixed-point tools — Prevent fixed-point tools from overriding Output data type**

*off (default) | on*

Select this parameter to prevent the fixed-point tools from overriding the **Output** data type you specify on the block. For more information, see “Use Lock Output Data Type Setting” (Fixed-Point Designer).

**Programmatic Use**

**Block Parameter:** LockScale

**Type:** character vector

**Values:** 'off' | 'on'

**Default:** 'off'

**Dimensions (-1 to infer from Initial value) — Dimensions of data store**

-1 (default) | scalar | vector | matrix

Dimensions of the data store. The default value, -1, enables you to set the dimensions of the data store by using the **Initial value** parameter. However, in this case, you cannot use scalar expansion with the initial value. You must specify the initial value by using an array that has the dimensions that you want.

If you use a value other than -1, specify the same dimensions as the dimensions of the **Initial value** parameter, unless you specify the initial value as a scalar (for scalar expansion) or a MATLAB structure. If the data store represents an array of buses, and if you use a MATLAB structure for the initial value, you can specify dimensions to initialize the array of buses with this structure.

**Programmatic Use**

**Block Parameter:** Dimensions

**Type:** character vector

**Values:** scalar | vector | matrix

**Default:** '-1'

**Interpret vector parameters as 1-D — Interpret vectors as 1-D**

on (default) | off

Specify that the data store interpret vector initial values as one-dimensional.
By default, MATLAB represents vector data as matrices, which have two dimensions. For example, MATLAB represents the vector \([1\ 2\ 3]\) as a 1-by-3 matrix.

When you select this parameter, the data store represents vector data by using only one dimension instead of two. For example, if you specify an initial value of \([1\ 2\ 3]\), the data store stores a one-dimensional vector with three elements.

For more information, see “Determining the Output Dimensions of Source Blocks”.

**Programmatic Use**  
**Block Parameter:** VectorParams1D  
**Type:** character vector  
**Values:** 'off' | 'on'  
**Default:** 'on'

**Signal type — Complexity of data store values**  
**auto** (default) | **real** | **complex**

Specify the numeric type, real or complex, of the values in the data store.

**Programmatic Use**  
**Block Parameter:** SignalType  
**Type:** character vector  
**Values:** 'auto' | 'real' | 'complex'  
**Default:** 'auto'

**Share across model instances — Allow Model blocks to read from the same data store**  
**off** (default) | **on**

In a single model reference hierarchy, when you use multiple Model blocks to refer to a model that contains a Data Store Memory block, by default, each instance of the referenced model (each Model block) reads from and writes to a separate copy of the data store. When you select **Share across model instances**, instead of interacting with a separate copy, all of the instances read from and write to the same data store.

When you set the model configuration parameter **Code interface packaging** to **Reusable function** to generate reentrant code from a model (Simulink Coder), a data store with **Share across model instances** selected appears in the code as a global symbol that the generated entry-point functions access directly. For example, a global symbol is a global variable or a field of a global structure variable. Therefore, each call that your code makes to the entry-point functions (each instance of the model) shares the data.
For an example, see “Share Data Store Between Instances of a Reusable Algorithm” on page 15-432. For more information, see “Share Data Among Referenced Model Instances”.

**Programmatic Use**

**Block Parameter:** ShareAcrossModelInstances  
**Type:** character vector  
**Values:** 'off' | 'on'  
**Default:** 'off'

**Data store name must resolve to Simulink signal object — Require data store name resolve to Simulink signal object**

`off` (default) | `on`

Specify that Simulink software, when compiling the model, searches the model and base workspace for a `Simulink.Signal` object having the same name, as described in “Symbol Resolution”. If Simulink does not find such an object, the compilation stops with an error. Otherwise, Simulink compares the attributes of the signal object to the corresponding attributes of the Data Store Memory block. If the block and the object attributes are inconsistent, Simulink halts model compilation and displays an error.

**Programmatic Use**

**Block Parameter:** StateMustResolveToSignalObject  
**Type:** character vector  
**Values:** 'off' | 'on'  
**Default:** 'off'

**Signal object class — Custom storage class package**

`Simulink.Signal` (default) | object of a class that is derived from `Simulink.Signal`

Choose a custom storage class package by selecting a signal object class that the target package defines. For example, to apply custom storage classes from the built-in package `mpt`, select `mpt.Signal`. Unless you use an ERT-based code generation target with Embedded Coder® software, custom storage classes do not affect the generated code.

For information about storage classes, see “Apply Storage Classes to Individual Signal, State, and Parameter Data Elements” (Simulink Coder). For information about custom storage classes, see “Apply Built-In and Customized Storage Classes to Data Elements” (Embedded Coder).

**Programmatic Use**

**Block Parameter:** StateSignalObject
Type: character vector  
Values: 'Simulink.Signal' | ...  
Default: 'Simulink.Signal'

Storage class — Storage class for code generation
Auto (default) | Model default | ExportedGlobal | ImportedExtern | ImportedExternPointer | BitField (Custom) | Volatile (Custom) | ExportToFile (Custom) | ImportFromFile (Custom) | FileScope (Custom) | AutoScope (Custom) | Struct (Custom) | GetSet (Custom) | Reusable (Custom)

Applies the storage class or custom storage class that you select from the list. For information about storage classes, see “Apply Storage Classes to Individual Signal, State, and Parameter Data Elements” (Simulink Coder). For information about custom storage classes, see “Apply Built-In and Customized Storage Classes to Data Elements” (Embedded Coder).

Use Signal object class to select custom storage classes from a package other than Simulink.

To programmatically set this parameter, use StateStorageClass or StateSignalObject. See “Apply Storage Classes to Individual Signal, State, and Parameter Data Elements” (Simulink Coder).

Programmatic Use
Block Parameter: StateStorageClass  
Type: character vector  
Values: 'Auto' | 'Model default' | 'ExportedGlobal' | 'ImportedExtern' | 'ImportedExternPointer' | 'Custom' | ...  
Default: 'Auto'

TypeQualifier — Storage type qualifier
'' (default) | const | volatile |...

Specify a storage type qualifier such as const or volatile.

Note TypeQualifier will be removed in a future release. To apply storage type qualifiers to data, use custom storage classes and memory sections. Unless you use an ERT-based code generation target with Embedded Coder, custom storage classes and memory sections do not affect the generated code.
During simulation, the block uses the following values:

- The initial value of the signal object to which the state name is resolved
- Minimum and Maximum values of the signal object

For more information, see “Data Objects”.

**Dependencies**

To enable this parameter, set **Code generation storage class** to ExportedGlobal, ImportedExtern, ImportedExternPointer, or Model default. This parameter is hidden unless you previously set its value.

**Programmatic Use**

**Block Parameter:** RTWStateStorageTypeQualifier  
**Type:** character vector  
**Values:** '' | 'const' | 'volatile' | ...  
**Default:** ''

**Diagnostics**

**Detect Read Before Write — Action when model attempts to read data before writing in current time step**  
warning (default) | none | error

Select the diagnostic action to take if the model attempts to read data from a data store to which it has not written data in this time step. See also the “Detect read before write” diagnostic in the **Data Store Memory block** section of the **Model Configuration Parameters > Diagnostics > Data Validity** pane.

- None — Produce no response.
- Warning — Display a warning and continue the simulation.
- Error — Terminate the simulation and display an error.

**Programmatic Use**

**Block Parameter:** ReadBeforeWriteMsg  
**Type:** character vector  
**Values:** 'none' | 'warning' | 'error'  
**Default:** 'warning'
**Detect Write After Read** — Action when block attempts to write after reading in same time step
warning (default) | none | error

Select the diagnostic action to take if the model attempts to write data to the data store after previously reading data from it in the current time step. See also the “Detect write after read” diagnostic in the Data Store Memory block section of the Model Configuration Parameters > Diagnostics > Data Validity pane.

- **None** — Produce no response.
- **Warning** — Display a warning and continue the simulation.
- **Error** — Terminate the simulation and display an error.

**Programmatic Use**

**Block Parameter:** WriteAfterReadMsg  
**Type:** character vector  
**Values:** 'none' | 'warning' | 'error'  
**Default:** 'warning'

**Detect Write After Write** — Action when model writes twice in same time step
warning (default) | none | error

Select the diagnostic action to take if the model attempts to write data to the data store twice in succession in the current time step. See also the “Detect write after write” diagnostic in the Data Store Memory block section of the Model Configuration Parameters > Diagnostics > Data Validity pane.

- **None** — Produce no response.
- **Warning** — Display a warning and continue the simulation.
- **Error** — Terminate the simulation and display an error.

**Programmatic Use**

**Block Parameter:** WriteAfterWriteMsg  
**Type:** character vector  
**Values:** 'none' | 'warning' | 'error'  
**Default:** 'warning'
Logging

Log data store data — Log data store data
off (default) | on

Select this option to save the values of this signal to the MATLAB workspace during simulation. See “Signal Logging” for details.

Programmatic Use
Block Parameter: DataLogging
Type: character vector
Values: 'off' | 'on'
Default: 'off'

Logging name — Name associated with logged signal data
Use data store name (default) | Custom

Use this pair of controls, consisting of a list box and an edit field, to specify the name associated with logged signal data.

Simulink uses the signal name as its logging name by default. To specify a custom logging name, select Custom from the list box and enter the custom name in the adjacent edit field.

Programmatic Use
Block Parameter: DataLoggingNameMode
Type: character vector
Values: 'SignalName' | 'Custom'
Default: ''

Note If you set DataLoggingNameMode to Custom, you must specify the name associated with logged signal data using the DataLoggingName parameter.

Block Parameter: DataLoggingName
Type: character vector
Values: character vector
Default: ''

Limit data points to last — Discard all but the last N data points
5000 | non-zero integer
Discard all but the last $N$ data points, where $N$ is the number that you enter in the adjacent edit field. For more information, see “Log Data Stores”.

**Programmatic Use**

**Block Parameter:** DataLoggingMaxPoints

**Type:** character vector

**Values:** nonzero integer

**Default:** '5000'

**Decimation — Log every $N$th data point**

2 (default) | integer

Log every $N$th data point, where $N$ is the number that you enter in the adjacent edit field. For example, suppose that your model uses a fixed-step solver with a step size of $0.1$ s. If you select this option and accept the default decimation value (2), Simulink records data points for this signal at times $0.0$, $0.2$, $0.4$, and so on. For more information, see “Log Data Stores”.

**Programmatic Use**

**Block Parameter:** DataLoggingLimitDataPoints

**Type:** character vector

**Values:** non-zero integer

**Default:** '2'

### Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>bus</th>
<th>double</th>
<th>enumerated</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
<th>string</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

PLC Code Generation
Generate Structured Text code using Simulink® PLC Coder™.

To generate PLC code for a model that uses a Data Store Memory block, first define a Simulink.Signal in the base workspace. Then in the Signal Attributes tab of the block parameters, set the data store name to resolve to that of the Simulink.Signal object.

Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

See Also
Data Store Read | Data Store Write

Topics
“Storing Data Using Data Store Memory Blocks”
“Data Stores”
“Choosing How to Store Global Data”
“Apply Storage Classes to Individual Signal, State, and Parameter Data Elements” (Simulink Coder)
“Apply Built-In and Customized Storage Classes to Data Elements” (Embedded Coder)
“Access Data Stores with Simulink Blocks”
“Log Data Stores”

Introduced before R2006a
Data Store Read

Read data from data store
Library: Simulink / Signal Routing

Description

The Data Store Read block copies data from the named data store to its output. More than one Data Store Read block can read from the same data store.

The data store from which the data is read is determined by the location of the Data Store Memory block or signal object that defines the data store. For more information, see “Data Stores” and Data Store Memory.

Obtaining correct results from data stores requires ensuring that data store reads and writes occur in the expected order. See “Order Data Store Access” and “Data Store Diagnostics” for details.

Ports

Output

Port_1 — Values from specified data store
scalar | vector | matrix | N-D array

Values from the specified data store, output with the same data type and dimensions as in the data store. The block supports both real and complex signals. You can choose whether to output the entire data store, or only selected elements from the data store.

You can use arrays of buses with a Data Store Read block. For details about defining and using an array of buses, see “Combine Buses into an Array of Buses”.
Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus

**Parameters**

**Arguments**

**Data store name — Name of data store from which this block reads**
A (default) | name of data store

Specifies the name of the data store from which this block reads data. The adjacent drop-down list provides the names of Data Store Memory blocks that exist at the same level in the model as the Data Store Read block or at higher levels. The list also includes all Simulink.Signal objects in the base and model workspaces. To change the name, select a name from the list or enter the name directly in the edit field.

When Simulink software compiles the model containing this block, Simulink searches the model upwards from this block's level for a Data Store Memory block having the specified data store name. If Simulink software does not find such a block, it searches the model workspace and the MATLAB workspace for a Simulink.Signal object having the same name. See “Symbol Resolution” for more information about the search path.

If Simulink finds the signal object, it creates a hidden Data Store Memory block at the model's root level having the properties specified by the signal object and an initial value of 0. If Simulink software finds neither the Data Store Memory block nor the signal object, it halts the compilation and displays an error.

**Programmatic Use**

**Block Parameter:** DataStoreName

**Type:** character vector

**Values:** data store name

**Default:** 'A'

**Data store memory block — Data Store Memory block name**
block path

This field lists the Data Store Memory block that initialized the store from which this block reads.
Data store write blocks — Corresponding Data Store Write blocks
block path

This field lists the path to all Data Store Write blocks with the same data store name as this block that are in the same (sub)system or in any subsystem below it in the model hierarchy. Click any entry in this list to highlight the corresponding block in your model.

Sample time — Sample time
-1 (default) | scalar | vector

The sample time, which controls when the block reads from the data store. A value of -1 indicates that the sample time is inherited. See “Specify Sample Time” for more information.

Programmatic Use
Block Parameter: SampleTime
Type: character vector
Values: scalar | vector
Default: ' -1'

Element Selection

Elements in the array (Signals in the bus) — Elements in the associated data store
no default

List of elements in the associated data store. For data stores containing arrays, you can read the whole data store, or you can specify one or more elements of the data store. For bus signals, lists the elements in the associated data store. The list displays the maximum dimensions for each element, in parentheses.

You can select an element and then use one of the following approaches:

- Click Select>> to display that element (and all its subelements) in the Selected element(s) list.
- Use the Specify element(s) to select edit box to specify the elements that you want to select for reading. Then click Select>>.

To refresh the display and reflect modifications to the array or bus used in the data store, click Refresh.
Dependencies

The prompt for this section (Elements in the array or Signals in the bus) depends on the type of data in the data store.

**Programmatic Use**

**Block Parameter:** DataStoreElements  
**Type:** character vector  
**Values:** pound-delimited list of elements (See “Specification using the command line”.)  
**Default:** ''

**Specify element(s) to select — MATLAB expression defining the elements to select**  
no default

Enter a MATLAB expression to define the specific element that you want to read. For example, for a data store named DSM that has maximum dimensions of [3, 5], you could enter expressions such as DSM(2, 4) or DSM([1 3], 2) in the edit box and then click Select>>.

To apply the element selection, click **OK** or **Apply**.

**Programmatic Use**

**Block Parameter:** DataStoreElements  
**Type:** character vector  
**Values:** pound-delimited list of elements (See “Specification using the command line”.)  
**Default:** ''

**Selected element(s) — List of selected elements**  
no default

Displays the elements that you select from the data store. The Data Store Read block icon displays a port for each element that you specify.

To change the order of bus or matrix elements in the list, select the element in the list and click **Up** or **Down**. Changing the order of the elements in the list changes the order of the ports. To remove an element, click **Remove**.

**Programmatic Use**

**Block Parameter:** DataStoreElements  
**Type:** character vector  
**Values:** pound-delimited list of elements (See “Specification using the command line”.)
Default: ' '  

Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>bus</th>
<th>double</th>
<th>enumerated</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
<th>string</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

See Also
Data Store Memory | Data Store Write

Topics
“Data Stores”
“Rename Data Stores”
“Order Data Store Access”
“Access Data Stores with Simulink Blocks”
“Data Store Diagnostics”

**Introduced before R2006a**
Data Store Write

Write data to data store
Library: Simulink / Signal Routing

Description

The Data Store Write block copies the value at its input to the named data store. Each write operation performed by a Data Store Write block writes over the data store, replacing the previous contents.

The data store to which this block writes is determined by the location of the Data Store Memory block or signal object that defines the data store. For more information, see “Data Stores” and Data Store Memory. The size of the data store is set by the signal object or the Data Store Memory block that defines and initializes the data store. Each Data Store Write block that writes to that data store must write the same amount of data.

More than one Data Store Write block can write to the same data store. However, if two Data Store Write blocks attempt to write to the same data store during the same simulation step, results are unpredictable.

Obtaining correct results from data stores requires ensuring that data store reads and writes occur in the expected order. For details, see “Order Data Store Access” and “Data Store Diagnostics”.

You can log the values of a local or global data store data variable for all the steps in a simulation. For details, see “Log Data Stores”.
Ports

Input

**Port 1 — Values to write to data store**
scalar | vector | matrix | N-D array

Values to write to the specified data store. The Data Store Write block accepts a real or complex signal.

You can use an array of buses with a Data Store Write block. For details about defining and using an array of buses, see “Combine Buses into an Array of Buses”.

To assign a subset of the bus or matrix elements to the associated data store, use the **Element Assignment** pane. The Data Store Write block icon reflects the elements that you specify. For details, see “Accessing Specific Bus and Matrix Elements”.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus

Parameters

Arguments

**Data store name — Name of data store from which this block writes**
A (default) | name of data store

Specifies the name of the data store to which this block writes data. The adjacent drop-down list provides the names of Data Store Memory blocks that exist at the same level in the model as the Data Store Write block or at higher levels. The drop-down list also includes all Simulink.Signal objects in the base and model workspaces. To change the name, select a name from the drop-down or enter the name directly in the edit field.

When Simulink software compiles the model containing this block, Simulink searches the model upwards from this block's level for a Data Store Memory block having the specified data store name. If Simulink does not find such a block, it searches the model workspace and the MATLAB workspace for a Simulink.Signal object having the same name. If Simulink finds neither the Data Store Memory block nor the signal object, it halts the
compilation and displays an error. See “Symbol Resolution” for more information about the search path.

If Simulink finds a signal object, it creates a hidden Data Store Memory block at the model's root level having the properties specified by the signal object and an initial value set to a matrix of zeros. The dimensions of that matrix are inherited from the Dimensions property of the signal object.

**Programmatic Use**

**Block Parameter:** DataStoreName  
**Type:** character vector  
**Values:** data store name  
**Default:** 'A'

**Data store memory block — Data Store Memory block name**

block path

This field lists the Data Store Memory block that initialized the store to which this block writes.

**Data store read blocks — Corresponding Data Store Read blocks**

block path

This field lists the path to all Data Store Read blocks with the same data store name as this block that are in the same (sub)system or in any subsystem below it in the model hierarchy. Click any entry in this list to highlight the corresponding block in your model.

**Sample time — Sample time**

-1 (default) | scalar | vector

The sample time, which controls when the block writes to the data store. A value of -1 indicates that the sample time is inherited. See “Specify Sample Time” for more information.

**Programmatic Use**

**Block Parameter:** SampleTime  
**Type:** character vector  
**Values:** scalar | vector  
**Default:** '-1'


**Element Assignment**

Elements in the array (Signals in the bus) — Elements in the associated data store
no default

List of elements in the associated data store. For data stores with arrays, you can write the whole data store, or you can assign one or more elements to the whole data store. For data stores with a bus data type, you can expand the tree to view the bus elements. The list displays the maximum dimensions for each element, in parentheses.

You can select an element and then use one of the following approaches:

- Click **Select>>** to display that element (and all its subelements) in the **Selected element(s)** list.
- Use the **Specify element(s) to select** edit box to specify the elements that you want to select for writing. Then click **Select>>**.

To refresh the display and reflect modifications to the array or bus used in the data store, click **Refresh**.

**Dependencies**

The prompt for this section (Elements in the array or Signals in the bus) depends on the type of data in the data store.

**Programmatic Use**

**Block Parameter:** DataStoreElements  
**Type:** character vector  
**Values:** pound-delimited list of elements (See “Specification using the command line”.)

**Specify element(s) to assign — MATLAB expression defining the elements to assign**

no default

Enter a MATLAB expression to define the specific element that you want to write. For example, for a data store named DSM that has maximum dimensions of [3, 5], you could enter expressions such as DSM(2, 4) or DSM([1 3], 2) in the edit box. Then click **Select>>**.

To apply the element selection, click **OK** or **Apply**.
Programmatic Use

Block Parameter: DataStoreElements
Type: character vector
Values: pound-delimited list of elements (See “Specification using the command line”.)
Default: ''

Assigned element(s) — List of selected elements
no default

Displays the elements that you selected for assignment. The Data Store Write block icon displays a port for each element that you specify.

To change the order of bus or matrix elements in the list, select the element in the list and click Up or Down. Changing the order of the elements in the list changes the order of the ports. To remove an element, click Remove.

Programmatic Use

Block Parameter: DataStoreElements
Type: character vector
Values: pound-delimited list of elements (See “Specification using the command line”.)
Default: ''

Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>bus</th>
<th>double</th>
<th>enumerated</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
<th>string</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

See Also
Data Store Memory | Data Store Read

Topics
“Data Stores”
“Rename Data Stores”
“Order Data Store Access”
“Access Data Stores with Simulink Blocks”
“Log Data Stores”
“Data Store Diagnostics”

Introduced before R2006a
Data Type Conversion

Convert input signal to specified data type

Library:
- Simulink / Commonly Used Blocks
- Simulink / Signal Attributes
- HDL Coder / Commonly Used Blocks
- HDL Coder / HDL Floating Point Operations
- HDL Coder / Signal Attributes

Description

The Data Type Conversion block converts an input signal of any Simulink data type to the data type that you specify.

Note
To control the output data type by specifying block parameters, or to inherit a data type from a downstream block, use the Data Type Conversion block. To inherit a data type from a different signal in the model, use the Data Type Conversion Inherited block.

Convert Fixed-Point Signals

When you convert between fixed-point data types, the **Input and output to have equal** parameter controls block behavior. This parameter does not change the behavior of the block when:

- The input and output do not have a fixed-point data type.
- The input or output has a fixed-point data type with trivial scaling.

For more information about fixed-point numbers, see “Fixed-Point Numbers in Simulink” (Fixed-Point Designer).

To convert a signal from one data type to another by attempting to preserve the real-world value of the input signal, select **Real World Value (RWV)**, the default setting. The block accounts for the limits imposed by the scaling of the input and output and attempts to generate an output of equal real-world value.

To change the real-world value of the input signal by performing a scaling reinterpretation of the stored integer value, select **Stored Integer (SI)**. Within the
limits of the specified data types, the block attempts to preserve the stored integer value of the signal during conversion. A best practice is to specify input and output data types using the same word length and signedness. Doing so ensures that the block changes only the scaling of the signal. Specifying a different signedness or word length for the input and output could produce unexpected results such as range loss or unexpected sign extensions. For an example, see “Convert Data Types in Simulink Models” on page 14-153.

If you select **Stored Integer (SI)**, the block does not perform a lower-level bit reinterpretation of a floating-point input signal. For example, if the input is `single` and has value 5, the bits that store the input in memory are given in hexadecimal by the following command.

```plaintext
num2hex(single(5))
```

```plaintext
40a00000
```

However, the Data Type Conversion block does not treat the stored integer value as `40a00000`, but instead as the real-world value, 5. After conversion, the stored integer value of the output is 5.

**Cast Enumerated Signals**

Use a Data Type Conversion block to cast enumerated signals as follows:

1. To cast a signal of enumerated type to a signal of any numeric type.
   
   The underlying integers of all enumerated values input to the Data Type Conversion block must be within the range of the numeric type. Otherwise, an error occurs during simulation.

2. To cast a signal of any integer type to a signal of enumerated type.

   The value input to the Data Type Conversion block must match the underlying value of an enumerated value. Otherwise, an error occurs during simulation.

   You can enable the **Saturate on integer overflow** parameter so that Simulink uses the default value of the enumerated type when the value input to the block does not match the underlying value of an enumerated value. See “Type Casting for Enumerations” (Simulink Coder).

You cannot use a Data Type Conversion block in these cases:
• To cast a noninteger numeric signal to an enumerated signal.
• To cast a complex signal to an enumerated signal, regardless of the data types of the real and imaginary parts of the complex signal.

See “Simulink Enumerations” for information on working with enumerated types.

**Ports**

**Input**

**Port_1 — Input signal**

scalar | vector | matrix | N-D array

Input signal, specified as a scalar, vector, matrix, or N-D array. The input can be any real- or complex-valued signal. If the input is real, the output is real. If the input is complex, the output is complex. The block converts the input signal to the **Output data type** you specify.

When you are converting fixed-point data types, use the **Input and output to have equal** parameter to determine whether the conversion happens based on the Real World Value (RWV) or Stored Integer (SI) value of the signal. For more information, see “Convert Fixed-Point Signals” on page 1-302.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated

**Output**

**Port_1 — Output signal**

scalar | vector | matrix | N-D array

Output signal, converted to the data type you specify, with the same dimensions as the input signal.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated
Parameters

Output minimum — Minimum output value for range checking

[ ] (default) | scalar

Lower value of the output range that Simulink checks.

Simulink uses the minimum to perform:

- Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters”) for some blocks.
- Simulation range checking (see “Signal Ranges” and “Enable Simulation Range Checking”).
- Automatic scaling of fixed-point data types.
- Optimization of the code that you generate from the model. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. For more information, see “Optimize using the specified minimum and maximum values” (Simulink Coder).

**Note** Output minimum does not saturate or clip the actual output signal. Use the Saturation block instead.

Programmatic Use

**Block Parameter:** OutMin

**Type:** character vector

**Values:** '[]' | scalar

**Default:** '[]'

Output maximum — Maximum output value for range checking

[ ] (default) | scalar

Upper value of the output range that Simulink checks.

Simulink uses the maximum value to perform:

- Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters”) for some blocks.
- Simulation range checking (see “Signal Ranges” and “Enable Simulation Range Checking”).
• Automatic scaling of fixed-point data types.

• Optimization of the code that you generate from the model. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. For more information, see “Optimize using the specified minimum and maximum values” (Simulink Coder).

**Note** Output maximum does not saturate or clip the actual output signal. Use the Saturation block instead.

**Programmatic Use**
**Block Parameter:** OutMax
**Type:** character vector
**Values:** '[' ']'| scalar
**Default:** '[' ']

**Output data type — Output data type**
Inherit: Inherit via back propagation (default) | double | single | int8 | uint8 | int16 | uint16 | int32 | uint32 | int64 | uint64 | boolean | fixdt(1,16) | fixdt(1,16,0) | fixdt(1,16,2^0,0) | Enum: <class name> | <data type expression>

Choose the data type for the output. The type can be inherited, specified directly, or expressed as a data type object such as Simulink.NumericType.

**Programmatic Use**
**Block Parameter:** OutDataTypeStr
**Type:** character vector
**Values:** 'Inherit: Inherit via back propagation' | 'double' | 'single' | 'int8' | 'uint8' | 'int16' | 'uint16' | 'int32' | 'uint32' | 'int64' | 'uint64' | 'fixdt(1,16)' | 'fixdt(1,16,0)' | 'fixdt(1,16,2^0,0)' | 'Enum: <class name>' | '<data type expression>'
**Default:** 'Inherit: Inherit via back propagation'

**Lock output data type setting against changes by the fixed-point tools — Prevent fixed-point tools from overriding Output data type**
off (default) | on

Select this parameter to prevent the fixed-point tools from overriding the Output data type you specify on the block. For more information, see “Use Lock Output Data Type Setting” (Fixed-Point Designer).
Programmatic Use
Block Parameter: LockScale
Type: character vector
Values: 'off' | 'on'
Default: 'off'

Input and output to have equal — Constraint for converting fixed-point data types
Real World Value (RWV) (default) | Stored Integer (SI)

Specify which type of input and output must be equal, in the context of fixed-point data representation.

- Real World Value (RWV) — Specifies the goal of making the Real World Value (RWV) of the input equal to the Real World Value (RWV) of the output.
- Stored Integer (SI) — Specifies the goal of making the Stored Integer (SI) value of the input equal to the Stored Integer (SI) value of the output.

Programmatic Use
Block Parameter: ConvertRealWorld
Type: character vector
Values: 'Real World Value (RWV)' | 'Stored Integer (SI)'
Default: 'Real World Value (RWV)'

Integer rounding mode — Specify the rounding mode for fixed-point operations
Floor (default) | Ceiling | Convergent | Nearest | Round | Simplest | Zero

Choose one of these rounding modes.

Ceiling
Rounds both positive and negative numbers toward positive infinity. Equivalent to the MATLAB ceil function.

Convergent
Rounds number to the nearest representable value. If a tie occurs, rounds to the nearest even integer. Equivalent to the Fixed-Point Designer convergent function.

Floor
Rounds both positive and negative numbers toward negative infinity. Equivalent to the MATLAB floor function.
Nearest
Rounds number to the nearest representable value. If a tie occurs, rounds toward positive infinity. Equivalent to the Fixed-Point Designer nearest function.

Round
Rounds number to the nearest representable value. If a tie occurs, rounds positive numbers toward positive infinity and rounds negative numbers toward negative infinity. Equivalent to the Fixed-Point Designer round function.

Simplest
Automatically chooses between round toward floor and round toward zero to generate rounding code that is as efficient as possible.

Zero
Rounds number toward zero. Equivalent to the MATLAB fix function.

Programmatic Use
Block Parameter: RndMeth
Type: character vector
Values: 'Ceiling' | 'Convergent' | 'Floor' | 'Nearest' | 'Round' | 'Simplest' | 'Zero'
Default: 'Floor'

See Also
For more information, see “Rounding” (Fixed-Point Designer).

Saturate on integer overflow — Method of overflow action
off (default) | on

Specify whether overflows saturate or wrap.

- off — Overflows wrap to the appropriate value that the data type can represent.
  For example, the number 130 does not fit in a signed 8-bit integer and wraps to -126.
- on — Overflows saturate to either the minimum or maximum value that the data type can represent.
  For example, an overflow associated with a signed 8-bit integer can saturate to -128 or 127.

Tip
• Consider selecting this check box when your model has a possible overflow and you want explicit saturation protection in the generated code.

• Consider clearing this check box when you want to optimize efficiency of your generated code.

  Clearing this check box also helps you to avoid overspecifying how a block handles out-of-range signals. For more information, see “Check for Signal Range Errors”.

• When you select this check box, saturation applies to every internal operation on the block, not just the output or result.

• In general, the code generation process can detect when overflow is not possible. In this case, the code generator does not produce saturation code.

---

**Programmatic Use**

**Block Parameter:** SaturateOnIntegerOverflow  
**Type:** character vector  
**Values:** 'off' | 'on'  
**Default:** 'off'

**Sample time — Specify sample time as a value other than -1**

-1 (default) | scalar | vector

Specify the sample time as a value other than -1. For more information, see “Specify Sample Time”.

**Dependencies**

This parameter is not visible unless it is explicitly set to a value other than -1. To learn more, see “Blocks for Which Sample Time Is Not Recommended”.

---

**Programmatic Use**

**Block Parameter:** SampleTime  
**Type:** character vector  
**Values:** scalar or vector  
**Default:** ' -1'
**Block Characteristics**

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>double</th>
<th>enumerated</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
</table>

| Direct Feedthrough    | yes     |        |            |             |         |        |
| Multidimensional      | yes     |        |            |             |         |        |
| Signals               |         |        |            |             |         |        |
| Variable-Size         | yes     |        |            |             |         |        |
| Signals               |         |        |            |             |         |        |
| Zero-Crossing Detection| no      |        |            |             |         |        |

**Extended Capabilities**

**C/C++ Code Generation**
Generate C and C++ code using Simulink® Coder™.

**HDL Code Generation**
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

**Note** If you use double data types in your model, use this block for conversion between double and single data types. You cannot use the block to convert between double and fixed-point data types.

**HDL Architecture**
This block has a single, default HDL architecture.
HDL Block Properties

<table>
<thead>
<tr>
<th>General</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ConstrainedOutputPipeline</strong></td>
</tr>
<tr>
<td><strong>InputPipeline</strong></td>
</tr>
<tr>
<td><strong>OutputPipeline</strong></td>
</tr>
</tbody>
</table>

Native Floating Point

With the HDL Model Checker, you can replace Data Type Conversion blocks that use the Stored Integer (SI) mode and convert between floating-point and fixed-point data types with Float Typecast blocks.

<table>
<thead>
<tr>
<th>LatencyStrategy</th>
<th>Specify whether to map the blocks in your design to inherit, Max, Min, Zero, or Custom for the floating-point operator. The default is inherit. See also “LatencyStrategy” (HDL Coder).</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>NFPCustomLatency</strong></td>
<td>To specify a value, set LatencyStrategy to Custom. HDL Coder adds latency equal to the value that you specify for the NFPCustomLatency setting. See also “NFPCustomLatency” (HDL Coder).</td>
</tr>
</tbody>
</table>

Complex Data Support

This block supports code generation for complex signals.

Restrictions

If you configure a Data Type Conversion block for double to fixed-point conversion or fixed-point to double conversion, a warning is displayed during code generation.
PLC Code Generation
Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

See Also
Data Type Conversion Inherited | Data Type Propagation | Data Type Scaling Strip

Topics
“Control Signal Data Types”
“About Data Types in Simulink”
“Simulink Enumerations”
“Fixed Point”

Introduced before R2006a
Data Type Conversion Inherited

Convert from one data type to another using inherited data type and scaling

Library: Simulink / Signal Attributes

Description

The Data Type Conversion Inherited block forces dissimilar data types to be the same. The first input is used as the reference signal. The second input, \( u \), is converted to the reference type by inheriting the data type and scaling information. (For a description of the port order for various block orientations, see “Port Location After Rotating or Flipping”.)

Inheriting the data type and scaling provides these advantages:

- It makes reusing existing models easier.
- It allows you to create new fixed-point models with less effort since you can avoid the detail of specifying the associated parameters.

Ports

Input

**Port_1 — Reference signal**

scalar | vector | matrix | N-D array

Reference signal, defining the data type used to convert input signal \( u \).

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated

**u — Input signal to convert**

scalar | vector | matrix | N-D array
Input signal to convert to the reference data type, specified as a scalar, vector, matrix, or N-D array.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 |
Boolean | fixed point | enumerated

Output

\( y \) — Output signal
scalar | vector | matrix | N-D array

Output is the input signal \( u \), converted to the reference data type.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 |
Boolean | fixed point | enumerated

Parameters

**Input and Output to have equal — Constraint for converting fixed-point data types**

Real World Value (RWV) (default) | Stored Integer (SI)

Specify which type of input and output must be equal, in the context of fixed-point data representation.

- **Real World Value (RWV)** — Specifies the goal of making the Real World Value (RWV) of the input equal to the Real World Value (RWV) of the output.
- **Stored Integer (SI)** — Specifies the goal of making the Stored Integer (SI) value of the input equal to the Stored Integer (SI) value of the output.

**Programmatic Use**

Block Parameter: ConvertRealWorld

Type: character vector

Values: 'Real World Value (RWV)' | 'Stored Integer (SI)'

Default: 'Real World Value (RWV)'

**Integer rounding mode — Rounding mode for fixed-point operations**

Floor (default) | Ceiling | Convergent | Nearest | Round | Simplest | Zero

Specify the rounding mode for fixed-point operations. For more information, see “Rounding” (Fixed-Point Designer).
**Programmatic Use**

**Block Parameter:** RndMeth  
**Type:** character vector  
**Values:** 'Ceiling' | 'Convergent' | 'Floor' | 'Nearest' | 'Round' | 'Simplest' | 'Zero'  
**Default:** 'Floor'

Saturate to max or min when overflows occur — Method of overflow action  
off (default) | on

When you select this check box, overflows saturate to the maximum or minimum value that the data type can represent. Otherwise, overflows wrap.

When you select this check box, saturation applies to every internal operation on the block, not just the output or result. In general, the code generation process can detect when overflow is not possible. In this case, the code generator does not produce saturation code.

**Programmatic Use**

**Block Parameter:** DoSatur  
**Type:** character vector  
**Values:** 'off' | 'on'  
**Default:** 'off'

**Block Characteristics**

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>double</th>
<th>enumerated</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

See Also
Data Type Conversion | Data Type Propagation

Topics
“Control Signal Data Types”
“About Data Types in Simulink”
“Fixed Point”

Introduced before R2006a
Data Type Duplicate

Force all inputs to same data type

Library: Simulink / Signal Attributes
        HDL Coder / Signal Attributes

Description

The Data Type Duplicate block forces all inputs to have the same data type. Other attributes of input signals, such as dimension, complexity, and sample time, are independent.

You can use the Data Type Duplicate block to check for consistency of data types among blocks. If all signals do not have the same data type, the block returns an error message.

The Data Type Duplicate block is typically used such that one signal to the block controls the data type for all other blocks. The other blocks are set to inherit their data types via backpropagation.

The block can also be useful in a user created library. These library blocks can be placed in any model, and the data type for all library blocks are configured according to the usage in the model. To create a library block with more complex data type rules than duplication, use the Data Type Propagation block.

Ports

Input

Port_1 — First input signal
scalar | vector | matrix | N-D array

First input signal, specified as a scalar, vector, matrix, or N-D array. If all signals do not have the same data type, the block returns an error message.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated
Port_N — Nth input signal
scalar | vector | matrix | N-D array

Nth input signal, specified as a scalar, vector, matrix, or N-D array. If all signals do not have the same data type, the block returns an error message.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated

Parameters

Number of input ports — Number of block inputs
2 (default) | real-valued positive integer

Specify the number of inputs to this block as a real-valued positive integer.

Programmatic Use

Block Parameter: NumInputPorts
Type: character vector
Values: real-valued positive integer
Default: '2'

Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>bus</th>
<th>double</th>
<th>enumerated</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
<th>string</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture
This block can be used to force inputs to have same data type in subsystems that generate code, but is not included in the hardware implementation.

PLC Code Generation
Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

See Also
Data Type Conversion | Data Type Propagation

Topics
“Control Signal Data Types”
“About Data Types in Simulink”
“Fixed Point”

Introduced before R2006a
Data Type Propagation

Set data type and scaling of propagated signal based on information from reference signals

Library: Simulink / Signal Attributes

Description

The Data Type Propagation block allows you to control the data type and scaling of signals in your model. You can use this block along with fixed-point blocks that have their Output data type parameter configured to Inherit: Inherit via back propagation.

The block has three inputs: Ref1 and Ref2 are the reference inputs, while the Prop input back-propagates the data type and scaling information gathered from the reference inputs. This information is then passed on to other fixed-point blocks.

The block provides many choices for propagating data type and scaling information. For example, you can use:

- The number of bits from the Ref1 reference signal or the number of bits from widest reference signal
- The range from the Ref2 reference signal or the range of the reference signal with the greatest range
- A bias of zero, regardless of the biases used by the reference signals
- The precision of the reference signal with the least precision

You specify how data type information is propagated using the Propagated data type parameter:

- If you select Specify via dialog, then you manually specify the data type via the Propagated data type edit field.
• If you select Inherit via propagation rule, then you must use the parameters described in “Parameters” on page 1-323.

You specify how scaling information is propagated using the Propagated scaling parameter:

• If you select Specify via dialog, then you manually specify the scaling via the Propagated scaling edit field.
• If you select Inherit via propagation rule, then you must use the parameters described in “Parameters” on page 1-323.

After you use the information from the reference signals, you can apply a second level of adjustments to the data type and scaling. To do so, use individual multiplicative and additive adjustments. This flexibility has various uses. For example, if you are targeting a DSP, then you can configure the block so that the number of bits associated with a multiply and accumulate (MAC) operation is twice as wide as the input signal, and has a specific number of guard bits added to it.

The Data Type Propagation block also provides a mechanism to force the computed number of bits to a useful value. For example, if you are targeting a 16-bit micro, then the target C compiler is likely to support sizes of only 8 bits, 16 bits, and 32 bits. The block forces these three choices to be used. For example, suppose that the block computes a data type size of 24 bits. Since 24 bits is not directly usable by the target chip, the signal is forced up to 32 bits, which is natively supported.

There is also a method for dealing with floating-point reference signals. This method makes it easier to create designs that are easily retargeted between fixed-point chips and floating-point chips.

The Data Type Propagation block allows you to set up libraries of useful subsystems that are properly configured based on the connected signals. Without this data type propagation process, subsystems from a library are unlikely to work as desired with most integer or fixed-point signals. Manual intervention would be required to configure the data type and scaling. In many situations, this block can eliminate the manual intervention.

**Precedence Rules**

The precedence of the dialog box parameters decreases from top to bottom. Also:
• Double-precision reference inputs have precedence over all other data types.
• Single-precision reference inputs have precedence over integer and fixed-point data types.
• Multiplicative adjustments are carried out before additive adjustments.
• The number of bits is determined before the precision or positive range is inherited from the reference inputs.
• PosRange is one bit higher than the exact maximum positive range of the signal.
• The computed number-of-bits are promoted to the smallest allowable value that is greater than or equal to the computation. If none exists, then the block returns an error.

**Ports**

**Input**

**Ref1 — First reference signal**

scalar | vector | matrix | N-D array

First reference signal, from which to gather data type and scaling information.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 |
Boolean | fixed point

**Ref2 — Second reference signal**

scalar | vector | matrix | N-D array

Second reference signal from which to gather data type and scaling information.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 |
Boolean | fixed point

**Prop — Propagated data type and scaling**

data type and scaling

Data type and scaling information, back-propagated to the model. After the block gathers data type and scaling information from the reference signals, you can apply a second level of adjustments to the data type and scaling. To do so, specify individual multiplicative and additive adjustments in the block dialog box.
Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

## Parameters

### Propagated Type

1. **Propagated data type — Mode of specifying propagated data type**
   *Inherit via propagation rule* (default) | *Specify via dialog*

Specify whether to propagate the data type via the dialog box, or inherit the data type from the reference signals.

#### Dependencies

Setting this parameter to *Specify via dialog* enables the 1.1. **Propagated data type** (e.g. `fixdt(1,16)`, `fixdt('single')`).

#### Programmatic Use

**Block Parameter:** PropDataTypeMode
**Type:** character vector
**Values:** 'Specify via dialog' | 'Inherit via propagation rule'
**Default:** 'Inherit via propagation rule'

1.1. **Propagated data type** (e.g. `fixdt(1,16)`, `fixdt('single')`) —

**Propagated data type**

`fixdt(1,16)` (default) | data type string

Specify the data type to propagate.

#### Dependencies

To enable this parameter, set 1. **Propagated data type** to *Specify via dialog*.

#### Programmatic Use

**Block Parameter:** PropDataTypeMode
**Type:** character vector
**Values:** 'Specify via dialog' | 'Inherit via propagation rule'
**Default:** 'Inherit via propagation rule'
1.1 If any reference input is double, output is — Output data type when a reference input is double
double (default) | single

Specify the output data type as single or double. This parameter makes it easier to create designs that are easily retargeted between fixed-point chips and floating-point chips.

Dependencies

To enable this parameter, set Propagated data type to Inherit via propagation rule.

Programmatic Use
Block Parameter: IfRefDouble
Type: character vector
Values: 'double' | 'single'
Default: 'double'

1.2 If any reference input is single, output is — Output data type when a reference input is single
single (default) | double

Specify the output data type as single or double. This parameter makes it easier to create designs that are easily retargeted between fixed-point chips and floating-point chips.

Dependencies

To enable this parameter, set Propagated data type to Inherit via propagation rule.

Programmatic Use
Block Parameter: IfRefSingle
Type: character vector
Values: 'double' | 'single'
Default: 'single'

1.3 Is-Signed — Signedness of propagated data type
IsSigned1 or IsSigned2 (default) | IsSigned1 | IsSigned2 | TRUE | FALSE

Specify the sign of Prop as one of the following values.
### Data Type Propagation

#### Parameter Value

<table>
<thead>
<tr>
<th>Parameter Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IsSigned1</td>
<td>Prop is a signed data type if Ref1 is a signed data type.</td>
</tr>
<tr>
<td>IsSigned2</td>
<td>Prop is a signed data type if Ref2 is a signed data type.</td>
</tr>
<tr>
<td>IsSigned1 or IsSigned2</td>
<td>Prop is a signed data type if either Ref1 or Ref2 are signed data types.</td>
</tr>
<tr>
<td>TRUE</td>
<td>Ref1 and Ref2 are ignored, and Prop is always a signed data type.</td>
</tr>
<tr>
<td>FALSE</td>
<td>Ref1 and Ref2 are ignored, and Prop is always an unsigned data type.</td>
</tr>
</tbody>
</table>

For example, if the Ref1 signal is ufix(16), the Ref2 signal is sfix(16), and the IsSigned parameter is IsSigned1 or IsSigned2, then Prop is forced to be a signed data type.

#### Dependencies

To enable this parameter, set Propagated data type to Inherit via propagation rule.

#### Programmatic Use

**Block Parameter:** IsSigned  
**Type:** character vector  
**Values:** 'IsSigned1' | 'IsSigned2' | 'IsSigned1 or IsSigned2' | 'TRUE' | 'FALSE'  
**Default:** 'IsSigned1 or IsSigned2'

**1.4.1 Number-of-bits: Base — Number of bits for the base of the propagated data type**

max([NumBits1 NumBits2]) (default) | NumBits1 | NumBits2 | min([NumBits1 NumBits2]) | NumBits1+NumBits2

Specify the number of bits used by Prop for the base data type as one of the following values.

<table>
<thead>
<tr>
<th>Parameter Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NumBits1</td>
<td>The number of bits for Prop is given by the number of bits for Ref1.</td>
</tr>
<tr>
<td>Parameter Value</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>NumBits2</td>
<td>The number of bits for Prop is given by the number of bits for Ref2.</td>
</tr>
<tr>
<td>max([NumBits1</td>
<td>The number of bits for Prop is given by the reference signal with largest</td>
</tr>
<tr>
<td>NumBits2])</td>
<td>number of bits.</td>
</tr>
<tr>
<td>min([NumBits1</td>
<td>The number of bits for Prop is given by the reference signal with smallest</td>
</tr>
<tr>
<td>NumBits2])</td>
<td>number of bits.</td>
</tr>
<tr>
<td>NumBits1+NumBits2</td>
<td>The number of bits for Prop is given by the sum of the reference signal</td>
</tr>
<tr>
<td></td>
<td>bits.</td>
</tr>
</tbody>
</table>

For more information about the base data type, refer to Targeting an Embedded Processor (Fixed-Point Designer).

**Dependencies**

To enable this parameter, set Propagated data type to Inherit via propagation rule.

**Programmatic Use**

**Block Parameter:** NumBitsBase  
**Type:** character vector  
**Values:** 'NumBits1' | 'NumBits2' | 'max([NumBits1 NumBits2])' | 'min([NumBits1 NumBits2])' | 'NumBits1+NumBits2'  
**Default:** 'max([NumBits1 NumBits2])'

### 1.4.2 Number-of-bits: Multiplicative adjustment — Number of bits for multiplicative adjustment of propagated data type

1 (default) | positive integer

Specify the number of bits used by Prop by including a multiplicative adjustment that uses a data type of double. For example, suppose that you want to guarantee that the number of bits associated with a multiply and accumulate (MAC) operation is twice as wide as the input signal. To do this, set this parameter to 2.

**Dependencies**

To enable this parameter, set Propagated data type to Inherit via propagation rule.

**Programmatic Use**

**Block Parameter:** NumBitsMult
Type: character vector
Values: positive integer
Default: '1'

1.4.3 Number-of-bits: Additive adjustment — Number of bits for additive adjustment of propagated data type
0 (default) | positive integer

Specify the number of bits used by Prop by including an additive adjustment that uses a data type of double. For example, if you are performing multiple additions during a MAC operation, the result could overflow. To prevent overflow, you can associate guard bits with the propagated data type. To associate four guard bits, you specify the value 4.

Dependencies
To enable this parameter, set Propagated data type to Inherit via propagation rule.

Programmatic Use
Block Parameter: NumBitsAdd
Type: character vector
Values: scalar
Default: '0'

1.4.4 Number-of-bits: Allowable final values — Allowable number of bits in propagated data type
'1:128' (default) | scalar or vector of positive integers

Force the computed number of bits used by Prop to a useful value. For example, if you are targeting a processor that supports only 8 bits, 16 bits, and 32 bits, then you configure this parameter to [8,16,32]. The block always propagates the smallest specified value that fits. If you want to allow all fixed-point data types, you would specify the value 1:128.

Dependencies
To enable this parameter, set Propagated data type to Inherit via propagation rule.

Programmatic Use
Block Parameter: NumBitsAllowFinal
Type: character vector
Values: scalar or vector of positive integers
**Default:** '1:128'

**Propagated Scaling**

2. **Propagated scaling — Propagated scaling mode**

Inherit via propagation rule (default) | Specify via dialog | Obtain via best precision

Choose to propagate the scaling via the dialog box, inherit the scaling from the reference signals, or calculate the scaling to obtain best precision.

**Programmatic Use**

**Block Parameter:** PropScalingMode
**Type:** character vector
**Values:** Inherit via propagation rule | Specify via dialog | Obtain via best precision
**Default:** Inherit via propagation rule

2.1. **Propagated scaling (Slope or [Slope Bias]) — Slope or slope and bias**

$2^{-10}$ | Slope | [Slope Bias]

Specify the scaling as either a slope or a slope and bias.

**Dependencies**

To enable this parameter, set **Propagated scaling** to Specify via dialog.

**Programmatic Use**

**Block Parameter:** PropScaling
**Type:** character vector
**Values:** Slope | [Slope Bias]
**Default:** '2^-10'

2.1.1. **Values used to determine best precision scaling — Values to constrain precision**

[5 -7] (default)

Specify any values to be used to constrain the precision, such as the upper and lower limits on the propagated input. Based on the data type, the block selects a scaling such that these values can be represented with no overflow error and minimum quantization error.
Dependencies

To enable this parameter, set Propagated scaling to Obtain via best precision.

Programmatic Use

**Block Parameter:** ValuesUsedBestPrec  
**Type:** character vector  
**Values:** vector of values  
**Default:** ' [5 -7 ]'  

### 2.1.1. Slope: Base — Slope for base of the propagated data type

\[
\min([\text{Slope}_1 \text{ Slope}_2]) \text{ (default)} \bigg| \text{Slope}_1 \bigg| \text{Slope}_2 \bigg| \min([\text{Slope}_1 \text{ Slope}_2]) \bigg| \\
\max([\text{Bias}_1 \text{ Bias}_2]) \bigg| \text{Slope}_1*\text{Slope}_2 \bigg| \text{Slope}_1/\text{Slope}_2 \bigg| \text{PosRange}_1 \bigg| \text{PosRange}_2 \bigg| \\
\max([\text{PosRange}_1 \text{ PosRange}_2]) \bigg| \min([\text{PosRange}_1 \text{ PosRange}_2]) \bigg| \\
\text{PosRange}_1*\text{PosRange}_2 \bigg| \text{PosRange}_1/\text{PosRange}_2
\]

Specify the slope used by Prop for the base data type as one of the following values.

<table>
<thead>
<tr>
<th>Parameter Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slope1</td>
<td>The slope of Prop is given by the slope of Ref1.</td>
</tr>
<tr>
<td>Slope2</td>
<td>The slope of Prop is given by the slope of Ref2.</td>
</tr>
<tr>
<td>\text{max}([\text{Slope}_1 \text{ Slope}_2])</td>
<td>The slope of Prop is given by the maximum slope of the reference signals.</td>
</tr>
<tr>
<td>\text{min}([\text{Slope}_1 \text{ Slope}_2])</td>
<td>The slope of Prop is given by the minimum slope of the reference signals.</td>
</tr>
<tr>
<td>\text{Slope}_1*\text{Slope}_2</td>
<td>The slope of Prop is given by the product of the reference signal slopes.</td>
</tr>
<tr>
<td>\text{Slope}_1/\text{Slope}_2</td>
<td>The slope of Prop is given by the ratio of the Ref1 slope to the Ref2 slope.</td>
</tr>
<tr>
<td>\text{PosRange}_1</td>
<td>The range of Prop is given by the range of Ref1.</td>
</tr>
<tr>
<td>\text{PosRange}_2</td>
<td>The range of Prop is given by the range of Ref2.</td>
</tr>
<tr>
<td>\text{max}([\text{PosRange}_1 \text{ PosRange}_2])</td>
<td>The range of Prop is given by the maximum range of the reference signals.</td>
</tr>
<tr>
<td>\text{min}([\text{PosRange}_1 \text{ PosRange}_2])</td>
<td>The range of Prop is given by the minimum range of the reference signals.</td>
</tr>
<tr>
<td>\text{PosRange}_1*\text{PosRange}_2</td>
<td>The range of Prop is given by the product of the reference signal ranges.</td>
</tr>
<tr>
<td>Parameter Value</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------------</td>
<td>------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>PosRange1/PosRange2</td>
<td>The range of Prop is given by the ratio of the Ref1 range to the Ref2 range.</td>
</tr>
</tbody>
</table>

You control the precision of Prop with Slope1 and Slope2, and you control the range of Prop with PosRange1 and PosRange2. Also, PosRange1 and PosRange2 are one bit higher than the maximum positive range of the associated reference signal.

**Dependencies**

To enable this parameter, set **Propagated scaling** to Inherit via propagation rule.

**Programmatic Use**

**Block Parameter:** SlopeBase  
**Type:** character vector  
**Values:** 'Slope1' | 'Slope2' | 'max([Slope1 Slope2])' | 'min([Slope1 Slope2])' | 'Slope1*Slope2' | 'Slope1/Slope2' | 'PosRange1' | 'PosRange2' | 'max([PosRange1 PosRange2])' | 'min([PosRange1 PosRange2])' | 'PosRange1*PosRange2' | 'PosRange1/PosRange2'  
**Default:** 'min([Slope1 Slope2])'

### 2.1.2. Slope: Multiplicative adjustment — Slope of multiplicative adjustment of propagated data type

1 (default) | scalar

Specify the slope used by Prop by including a multiplicative adjustment that uses a data type of double. For example, if you want 3 bits of additional precision (with a corresponding decrease in range), the multiplicative adjustment is $2^{-3}$.

**Dependencies**

To enable this parameter, set **Propagated scaling** to Inherit via propagation rule.

**Programmatic Use**

**Block Parameter:** SlopeMult  
**Type:** character vector  
**Values:** scalar  
**Default:** '1'
**2.1.3. Slope: Additive adjustment — Slope of additive adjustment of propagated data type**

0 (default) | scalar

Specify the slope used by Prop by including an additive adjustment that uses a data type of double. An additive slope adjustment is often not needed. The most likely use is to set the multiplicative adjustment to 0, and set the additive adjustment to force the final slope to a specified value.

**Dependencies**

To enable this parameter, set Propagated scaling to Inherit via propagation rule.

**Programmatic Use**

Block Parameter: SlopeAdd
Type: character vector
Values: scalar
Default: '0'

**2.2.1. Bias: Base — Base bias for Prop**

Bias1 (default) | Bias2 | max([Bias1 Bias2]) | min([Bias1 Bias2]) | Bias1*Bias2 | Bias1/Bias2 | Bias1+Bias2 | Bias1-Bias2

Specify the bias used by Prop for the base data type. The parameter values are described as follows:

<table>
<thead>
<tr>
<th>Parameter Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bias1</td>
<td>The bias of Prop is given by the bias of Ref1.</td>
</tr>
<tr>
<td>Bias2</td>
<td>The bias of Prop is given by the bias of Ref2.</td>
</tr>
<tr>
<td>max([Bias1 Bias2])</td>
<td>The bias of Prop is given by the maximum bias of the reference signals.</td>
</tr>
<tr>
<td>min([Bias1 Bias2])</td>
<td>The bias of Prop is given by the minimum bias of the reference signals.</td>
</tr>
<tr>
<td>Bias1*Bias2</td>
<td>The bias of Prop is given by the product of the reference signal biases.</td>
</tr>
<tr>
<td>Bias1/Bias2</td>
<td>The bias of Prop is given by the ratio of the Ref1 bias to the Ref2 bias.</td>
</tr>
<tr>
<td>Bias1+Bias2</td>
<td>The bias of Prop is given by the sum of the reference biases.</td>
</tr>
<tr>
<td>Parameter Value</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------</td>
<td>-------------</td>
</tr>
<tr>
<td>Bias1-Bias2</td>
<td>The bias of Prop is given by the difference of the reference biases.</td>
</tr>
</tbody>
</table>

**Dependencies**

To enable this parameter, set **Propagated scaling** to Inherit via propagation rule.

**Programmatic Use**

**Block Parameter:** BiasBase  
**Type:** character vector  
**Values:** 'Bias1' | 'Bias2' | 'max([Bias1 Bias2])' | 'min([Bias1 Bias2])' | 'Bias1*Bias2' | 'Bias1/Bias2' | 'Bias1+Bias2' | 'Bias1-Bias2'  
**Default:** 'Bias1'

2.2.2. Bias: Multiplicative adjustment — Multiplicative bias for propagated data type

Specify the bias used by Prop by including a multiplicative adjustment that uses a data type of double.

This parameter is visible only when you set **Propagated scaling** to Inherit via propagation rule.

**Programmatic Use**

**Block Parameter:** BiasMult  
**Type:** character vector  
**Values:** scalar  
**Default:** '0'

2.3.2. Bias: Additive adjustment — Additive bias for propagated data type

Specify the bias used by Prop by including an additive adjustment that uses a data type of double.

If you want to guarantee that the bias associated with Prop is zero, configure both the multiplicative adjustment and the additive adjustment to 0.
Dependencies
To enable this parameter, set **Propagated scaling** to **Inherit via propagation rule**.

Programmatic Use
**Block Parameter:** BiasAdd  
**Type:** character vector  
**Values:** scalar  
**Default:** '0'

Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>double</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Extended Capabilities

**C/C++ Code Generation**
Generate C and C++ code using Simulink® Coder™.

**HDL Code Generation**
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.
HDL Architecture

This block can be used to control scaling and data types of signals that are part of subsystems that generate HDL code, but is not included in the hardware implementation.

Fixed-Point Conversion

Design and simulate fixed-point systems using Fixed-Point Designer™.

See Also

Data Type Conversion | Data Type Conversion Inherited | Data Type Duplicate

Topics

“Control Signal Data Types”
“About Data Types in Simulink”
“Fixed Point”

Introduced before R2006a
Data Type Scaling Strip

Remove scaling and map to built in integer

Library: Simulink / Signal Attributes

Description

The Data Type Scaling Strip block strips the scaling off a fixed-point signal. It maps the input data type to the smallest built-in data type that has enough data bits to hold the input. The stored integer value of the input is the value of the output. The output always has nominal scaling (slope = 1.0 and bias = 0.0), so the output does not distinguish between real world value and stored integer value.

Ports

Input

Port_1 — Input signal
scalar | vector | matrix

Input signal, specified as a scalar, vector, or matrix. The block strips the scaling off a fixed-point input signal, and outputs the stored integer value with the smallest possible built-in data type.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point

Output

Port_1 — Input signal mapped to built-in data type
scalar | vector | matrix
Stored integer value of the input signal with the smallest possible built-in data type, and the same dimensions as the input.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32

**Block Characteristics**

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>double</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Extended Capabilities**

**C/C++ Code Generation**
Generate C and C++ code using Simulink® Coder™.

**HDL Code Generation**
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

**Fixed-Point Conversion**
Design and simulate fixed-point systems using Fixed-Point Designer™.
See Also
Data Type Conversion | Data Type Duplicate | Data Type Propagation

Topics
“About Data Types in Simulink”
“Fixed Point”

Introduced before R2006a
Dead Zone

Provide region of zero output

Library: Simulink / Discontinuities
HDL Coder / Discontinuities

Description

The Dead Zone block generates zero output within a specified region, called its dead zone. You specify the lower limit (LL) and upper limit (UL) of the dead zone as the Start of dead zone and End of dead zone parameters. The block output depends on the input (U) and the values for the lower and upper limits.

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>U &gt;= LL and U &lt;= UL</td>
<td>Zero</td>
</tr>
<tr>
<td>U &gt; UL</td>
<td>U - UL</td>
</tr>
<tr>
<td>U &lt; LL</td>
<td>U - LL</td>
</tr>
</tbody>
</table>

Ports

Input

Port 1 — Input signal

Input signal to the dead-zone algorithm.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point
Output

Port_1 — Output signal
scalar | vector

Output signal after the dead-zone algorithm is applied to the input signal.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

Parameters

Start of dead zone — Specify the lower bound of the dead zone
'-0.5' (default) | scalar | vector

Specify dead zone lower limit. Set the value for Start of dead zone less than or equal to End of dead zone. When the input value is less than Start of dead zone, then the block shifts the output value down by the Start of dead zone value.

Programmatic Use
Block Parameter: LowerValue
Type: character vector
Value: scalar or vector less than or equal to UpperValue.
Default: '-0.5'

End of dead zone — Specify the upper limit of the dead zone
'0.5' (default) | scalar | vector

Specify dead zone upper limit. Set the value for End of dead zone greater than or equal to Start of dead zone. When the input value is greater than End of dead zone, then the block shifts the output value down by the End of dead zone value.

Programmatic Use
Block Parameter: UpperValue
Type: character vector
Value: scalar or vector greater than or equal to LowerValue.
Default: '0.5'

Saturate on integer overflow — Choose the behavior when integer overflow occurs
on (default) | boolean
<table>
<thead>
<tr>
<th>Action</th>
<th>Reasons for Taking This Action</th>
<th>What Happens for Overflows</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Select this check box.</td>
<td>Your model has possible overflow, and you want explicit saturation protection in the generated code.</td>
<td>Overflows saturate to either the minimum or maximum value that the data type can represent.</td>
<td>The maximum value that the <code>int8</code> (signed, 8-bit integer) data type can represent is 127. Any block operation result greater than this maximum value causes overflow of the 8-bit integer. With the check box selected, the block output saturates at 127. Similarly, the block output saturates at a minimum output value of -128.</td>
</tr>
<tr>
<td>Do not select this check box.</td>
<td>You want to optimize efficiency of your generated code.</td>
<td>Overflows wrap to the appropriate value that is representable by the data type.</td>
<td>The maximum value that the <code>int8</code> (signed, 8-bit integer) data type can represent is 127. Any block operation result greater than this maximum value causes overflow of the 8-bit integer. With the check box cleared, the software interprets the overflow-causing value as <code>int8</code>, which can produce an unintended result. For example, a block result of 130 (binary 1000 0010) expressed as <code>int8</code>, is -126.</td>
</tr>
</tbody>
</table>

When you select this check box, saturation applies to every internal operation on the block, not just the output or result. Usually, the code generation process can detect when overflow is not possible. In this case, the code generator does not produce saturation code.
Programmatic Use
Block Parameter: SaturateOnIntegerOverflow
Type: character vector
Value: 'off' | 'on'
Default: 'on'

Treat as gain when linearizing — Specify the gain value
0n (default) | boolean

The linearization commands in Simulink software treat this block as a gain in state space. Select this check box to cause the commands to treat the gain as 1. Clear the box to have the commands treat the gain as 0.

Programmatic Use
Block Parameter: LinearizeAsGain
Type: character vector
Value: 'off' | 'on'
Default: 'on'

Enable zero-crossing detection — Enable zero-crossing detection
on (default) | Boolean

Select to enable zero-crossing detection. For more information, see “Zero-Crossing Detection” in the Simulink documentation.

Programmatic Use
Block Parameter: ZeroCross
Type: character vector, string
Values: 'off' | 'on'
Default: 'on'

Sample time — Specify sample time as a value other than -1
-1 (default) | scalar | vector

Specify the sample time as a value other than -1. For more information, see “Specify Sample Time”.

Dependencies
This parameter is not visible unless it is explicitly set to a value other than -1. To learn more, see “Blocks for Which Sample Time Is Not Recommended”.

1-341
Programmatic Use
Block Parameter: SampleTime
Type: character vector
Values: scalar or vector
Default: '-1'

Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>double</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture
This block has a single, default HDL architecture.
### HDL Block Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ConstrainedOutputPipeline</strong></td>
<td>Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. For more details, see “ConstrainedOutputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td><strong>InputPipeline</strong></td>
<td>Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “InputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td><strong>OutputPipeline</strong></td>
<td>Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “OutputPipeline” (HDL Coder).</td>
</tr>
</tbody>
</table>

### Fixed-Point Conversion

Design and simulate fixed-point systems using Fixed-Point Designer™.

### See Also

Backlash | Dead Zone Dynamic

### Introduced before R2006a
Dead Zone Dynamic

Provide dynamic region of zero output

Library: Simulink / Discontinuities
HDL Coder / Discontinuities

Description

The Dead Zone Dynamic block generates a region of zero output based on dynamic input signals that specify the upper and lower limit. The block output depends on the input $u$, and the values of the input signals $u_{up}$ and $u_{lo}$.

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>$u \geq u_{lo}$ and $u \leq u_{up}$</td>
<td>Zero</td>
</tr>
<tr>
<td>$u &gt; u_{up}$</td>
<td>$u - u_{up}$</td>
</tr>
<tr>
<td>$u &lt; u_{lo}$</td>
<td>$u - u_{lo}$</td>
</tr>
</tbody>
</table>

The Dead Zone Dynamic block is a masked subsystem and does not have any parameters.

Ports

Input

$u$ — Input signal
scalar | vector

Input signal to the dead zone algorithm.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

$lo$ — Lower limit for the dead zone
scalar | vector
Dynamic value providing the lower bound of the region of zero output. When the input is less than \( \text{lo} \) then the output value is shifted down by value of \( \text{lo} \).

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

**up — Upper limit for the dead zone**

scalar

Dynamic value providing the upper bound of the region of zero output. When the input is greater than \( \text{up} \) then the output value is shifted down by value of \( \text{up} \).

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

**Output**

\( y \) — Output signal

scalar | vector

Output signal after the dynamic dead zone algorithm is applied to the input signal.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

**Block Characteristics**

<table>
<thead>
<tr>
<th>Data Types</th>
<th>double</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture
This block has a single, default HDL architecture.

HDL Block Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConstrainedOutputPipeline</td>
<td>Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. For more details, see “ConstrainedOutputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td>InputPipeline</td>
<td>Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “InputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td>OutputPipeline</td>
<td>Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “OutputPipeline” (HDL Coder).</td>
</tr>
</tbody>
</table>

PLC Code Generation
Generate Structured Text code using Simulink® PLC Coder™.
Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

See Also
Dead Zone | Triggered Subsystem

Introduced before R2006a
Decrement Real World

Decrease real-world value of signal by one

**Library:** Simulink / Additional Math & Discrete / Additional Math: Increment - Decrement
HDL Coder / Math Operations

**Description**

The Decrement Real World block decreases the real-world value of the signal by one. Overflows always wrap.

**Ports**

**Input**

**Port_1 — Input signal**
scalar | vector | matrix

Input signal, specified as a scalar, vector, or matrix.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

**Output**

**Port_1 — Output signal**
scalar | vector | matrix

Output is the real-world value of the input signal decreased by one. Overflows always wrap. The output has the same data type and dimensions as the input.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point
Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>double</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

The code generator does not explicitly group primitive blocks that constitute a nonatomic masked subsystem block in the generated code. This flexibility allows for more efficient code generation. In certain cases, you can achieve grouping by configuring the masked subsystem block to execute as an atomic unit by selecting the Treat as atomic unit option.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture
This block has a single, default HDL architecture.
HDL Block Properties

<table>
<thead>
<tr>
<th><strong>ConstrainedOutputPipeline</strong></th>
<th>Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. For more details, see “ConstrainedOutputPipeline” (HDL Coder).</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>InputPipeline</strong></td>
<td>Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “InputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td><strong>OutputPipeline</strong></td>
<td>Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “OutputPipeline” (HDL Coder).</td>
</tr>
</tbody>
</table>

**Fixed-Point Conversion**
Design and simulate fixed-point systems using Fixed-Point Designer™.

**See Also**
Decrement Stored Integer | Decrement Time To Zero | Decrement To Zero | Increment
Real World

**Topics**
“Fixed-Point Numbers”

**Introduced before R2006a**
Decrement Stored Integer

Decrease stored integer value of signal by one

**Library:** Simulink / Additional Math & Discrete / Additional Math: Increment - Decrement
HDL Coder / Math Operations

**Description**

The Decrement Stored Integer block decreases the stored integer value of a signal by one.

Floating-point signals also decrease by one, and overflows always wrap.

**Ports**

**Input**

*Port_1 — Input signal*

scalar | vector | matrix

Input signal, specified as a scalar, vector, or matrix.

**Data Types:** single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

**Output**

*Port_1 — Output signal*

scalar | vector | matrix

Output is the stored integer value of the input signal decreased by one. Floating-point signals also decrease by one, and overflows always wrap. The output has the same data type and dimensions as the input.
Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed_point

## Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>double</th>
<th>fixed_point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

## Extended Capabilities

### C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

The code generator does not explicitly group primitive blocks that constitute a nonatomic masked subsystem block in the generated code. This flexibility allows for more efficient code generation. In certain cases, you can achieve grouping by configuring the masked subsystem block to execute as an atomic unit by selecting the Treat as atomic unit option.

### HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.
HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConstrainedOutput Pipeline</td>
<td>Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. For more details, see “ConstrainedOutputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td>InputPipeline</td>
<td>Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “InputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td>OutputPipeline</td>
<td>Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “OutputPipeline” (HDL Coder).</td>
</tr>
</tbody>
</table>

Fixed-Point Conversion

Design and simulate fixed-point systems using Fixed-Point Designer™.

See Also

Decrement Real World | Decrement Time To Zero | Decrement To Zero | Increment Stored Integer

Topics

“Fixed-Point Numbers”

Introduced before R2006a
Decrement Time To Zero

Decrease real-world value of signal by sample time, but only to zero

Library: Simulink / Additional Math & Discrete / Additional Math: Increment - Decrement

Description

The Decrement Time To Zero block decreases the real-world value of the signal by the sample time, \( T_s \). The output never goes below zero.

Limitations

The Decrement Time To Zero block works only with fixed sample rates and does not work inside a triggered subsystem.

Ports

Input

Port 1 — Input signal
scalar | vector | matrix

Input signal, specified as a scalar, vector, or matrix.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

Output

Port 1 — Output signal
scalar | vector | matrix
Output is the real-world value of the input signal decreased by the sample time, $T_s$. The output never goes below zero. The output has the same data type and dimensions as the input.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

### Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>double</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Extended Capabilities

#### C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

#### Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

### See Also
Decrement Real World | Decrement Stored Integer | Decrement To Zero
Topics
“Fixed-Point Numbers”

Introduced before R2006a
Decrement To Zero

Decrease real-world value of signal by one, but only to zero

Library: Simulink / Additional Math & Discrete / Additional Math: Increment - Decrement

Description

The Decrement To Zero block decreases the real-world value of the signal by one. The output never goes below zero.

Ports

Input

Port_1 — Input signal
scalar | vector | matrix

Input signal, specified as a scalar, vector, or matrix.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

Output

Port_1 — Output signal
scalar | vector | matrix

Output is the real-world value of the input signal decreased by one. The output never goes below zero. The output has the same data type and dimensions as the input.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point
Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>double</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

The code generator does not explicitly group primitive blocks that constitute a nonatomic masked subsystem block in the generated code. This flexibility allows for more efficient code generation. In certain cases, you can achieve grouping by configuring the masked subsystem block to execute as an atomic unit by selecting the Treat as atomic unit option.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture
This block has a single, default HDL architecture.
### HDL Block Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConstrainedOutput</td>
<td>Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. For more details, see “ConstrainedOutputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td>InputPipeline</td>
<td>Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “InputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td>OutputPipeline</td>
<td>Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “OutputPipeline” (HDL Coder).</td>
</tr>
</tbody>
</table>

### Fixed-Point Conversion

Design and simulate fixed-point systems using Fixed-Point Designer™.

### See Also

Decrement Real World | Decrement Stored Integer | Decrement Time To Zero

### Topics

“Fixed-Point Numbers”

### Introduced before R2006a
Delay

Delay input signal by fixed or variable sample periods

**Library:**
- Simulink / Commonly Used Blocks
- Simulink / Discrete
- HDL Coder / Commonly Used Blocks
- HDL Coder / Discrete

**Description**

The Delay block outputs the input of the block after a delay. The block determines the delay time based on the value of the **Delay length** parameter. The block supports:

- Variable delay length
- Specification of the initial condition from an input port
- State storage
- Using a circular buffer instead of an array buffer for state storage
- Resetting the state to the initial condition with an external reset signal
- Controlling execution of the block at every time step with an external enable signal

The initial block output depends on several factors such as the **Initial condition** parameter and the simulation start time. For more information, see “Initial Block Output” on page 1-360. The **External reset** parameter determines if the block output resets to the initial condition on triggering. The **Show enable port** parameter determines if the block execution is controlled in every time step by an external enable signal.

**Initial Block Output**

The output in the first few time steps of the simulation depends on the block sample time, the delay length, and the simulation start time. The block supports specifying or inheriting discrete sample times to determine the time interval between samples. For more information, see “Specify Sample Time”.

The table shows the Delay block output for the first few time steps with these settings. The block inherits a discrete sample time as \[Tsampling, Toffset\], where \(Tsampling\)
is the sampling period and $T_{offset}$ is the initial time offset. $n$ is the value of the Delay length parameter and $T_{start}$ is the simulation start time for the model.

<table>
<thead>
<tr>
<th>Simulation Time Range</th>
<th>Block Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(T_{start})$ to $(T_{start} + T_{offset})$</td>
<td>Zero</td>
</tr>
<tr>
<td>$(T_{start} + T_{offset})$ to $(T_{start} + T_{offset} + n \times T_{sampling})$</td>
<td>Initial condition parameter</td>
</tr>
<tr>
<td>After $(T_{start} + T_{offset} + n \times T_{sampling})$</td>
<td>Input signal</td>
</tr>
</tbody>
</table>

### Variable-Size Support

The Delay block provides the following support for variable-size signals:

- The data input port $u$ accepts variable-size signals. The other input ports do not accept variable-size signals.
- The output port has the same signal dimensions as the data input port $u$ for variable-size inputs.

The rules that apply to variable-size signals depend on the input processing mode of the Delay block.

<table>
<thead>
<tr>
<th>Input Processing Mode</th>
<th>Rules for Variable-Size Signal Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Elements as channels (sample based)</td>
<td>• The signal dimensions change only during state reset when the block is enabled.</td>
</tr>
<tr>
<td></td>
<td>• The initial condition must be scalar.</td>
</tr>
<tr>
<td>Columns as channels (frame based)</td>
<td>• No support</td>
</tr>
</tbody>
</table>

### Bus Support

The Delay block provides the following support for bus signals:

- The data input port $u$ accepts virtual and nonvirtual bus signals. The other input ports do not accept bus signals.
- The output port has the same bus type as the data input port $u$ for bus inputs.
- Buses work with:
To use a bus signal as the input to a Delay block, specify the initial condition on the dialog box. The initial condition cannot come from the input port \( x_0 \). Support for virtual and nonvirtual buses depends on the initial condition that you specify and whether the **State name** parameter is empty or not.

<table>
<thead>
<tr>
<th>Initial Condition</th>
<th>State Name</th>
<th>Empty</th>
<th>Not Empty</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zero</td>
<td>Virtual and nonvirtual bus support</td>
<td>Virtual and nonvirtual bus support</td>
<td>Nonvirtual bus support only</td>
</tr>
<tr>
<td>Nonzero scalar</td>
<td>Virtual and nonvirtual bus support</td>
<td>No bus support</td>
<td>Non virtual bus support only</td>
</tr>
<tr>
<td>Nonscalar</td>
<td>No bus support</td>
<td>No bus support</td>
<td>No bus support</td>
</tr>
<tr>
<td>Structure</td>
<td>Virtual and nonvirtual bus support</td>
<td>Virtual and nonvirtual bus support</td>
<td>Nonvirtual bus support only</td>
</tr>
<tr>
<td>Partial structure</td>
<td>Virtual and nonvirtual bus support</td>
<td>Virtual and nonvirtual bus support</td>
<td>Nonvirtual bus support only</td>
</tr>
</tbody>
</table>

**Ports**

**Input**

\( u \) — **Data input signal**

* scalar | vector

Input data signal delayed according to parameters settings.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus

\( d \) — **Delay length**

* scalar
Delay length specified as inherited from an input port. Enabled when you select the **Delay length: Source** parameter as **Input port**.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | fixed point

**Enable — External enable signal**

*scalar*

Enable signal that enables or disables execution of the block. To create this port, select the **Show enable port** parameter.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | Boolean | fixed point

**External reset — External reset signal**

*scalar*

External signal that resets execution of the block to the initial condition. To create this port, select the **External reset** parameter.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | Boolean | fixed point

**x0 — Initial condition**

*scalar | vector*

Initial condition specified as inherited from an input port. Enabled when you select the **Initial Condition: Source** parameter as **Input port**.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | fixed point

**Output**

**Port_1 — Output signal**

*scalar | vector*

Output signal that is the input signal delayed by the length of time specified by the parameter **Delay length**. The initial value of the output signal depends on several conditions. See “Initial Block Output” on page 1-360.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | Boolean | fixed point | enumerated | bus
Parameters

Main

**Delay length — Delay length**
Dialog (default) | Input port

Specify whether to enter the delay length directly on the dialog box (fixed delay) or to inherit the delay from an input port (variable delay).

- If you set **Source** to **Dialog**, enter the delay length in the edit field under **Value**.
- If you set **Source** to **Input port**, verify that an upstream signal supplies a delay length for the d input port. You can also specify its maximum value by specifying the parameter **Upper limit**.

Specify the scalar delay length as a real, non-negative integer. An out-of-range or non-integer value in the dialog box (fixed delay) returns an error. An out-of-range value from an input port (variable delay) casts it into the range. A noninteger value from an input port (variable delay) truncates it to the integer.

**Programmatic Use**

**Block Parameter:** DelayLengthSource
**Type:** character vector
**Values:** 'Dialog' | 'Input port'
**Default:** 'Dialog'

**Block Parameter:** DelayLength
**Type:** character vector
**Values:** scalar
**Default:** '2'

**Block Parameter:** DelayLengthUpperLimit
**Type:** character vector
**Values:** scalar
**Default:** '100'

**Initial condition — Initial condition**
Dialog (default) | Input port

Specify whether to enter the initial condition directly on the dialog box or to inherit the initial condition from an input port.
• If you set **Source** to **Dialog**, enter the initial condition in the edit field under **Value**.
• If you set **Source** to **Input port**, verify that an upstream signal supplies an initial condition for the x<sub>0</sub> input port.

Simulink converts offline the data type of **Initial condition** to the data type of the input signal u using a round-to-nearest operation and saturation.

**Note** When **State name must resolve to Simulink signal object** is selected on the **State Attributes** pane, the block copies the initial value of the signal object to the **Initial condition** parameter. However, when the source for **Initial condition** is **Input port**, the block ignores the initial value of the signal object.

**Programmatic Use**
**Block Parameter:** InitialConditionSource
**Type:** character vector
**Values:** 'Dialog' | 'Input port'
**Default:** 'Dialog'

**Block Parameter:** InitialCondition
**Type:** character vector
**Values:** scalar
**Default:** '0.0'

**Input processing — Specify sample- or frame-based processing**
**Elements as channels (sample based) (default) | Columns as channels (frame based)**

Specify whether the block performs sample- or frame-based processing:

• **Columns as channels (frame based)** — Treat each column of the input as a separate channel (frame-based processing).

**Note** Frame-based processing requires a DSP System Toolbox license.

For more information, see “Sample- and Frame-Based Concepts” (DSP System Toolbox).

• **Elements as channels (sample based)** — Treat each element of the input as a separate channel (sample-based processing).
Use **Input processing** to specify whether the block performs sample- or frame-based processing. The block accepts frame-based signals for the input $u$. All other input signals must be sample-based.

<table>
<thead>
<tr>
<th>Input Signal $u$</th>
<th>Input Processing Mode</th>
<th>Block Works?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sample based</td>
<td>Sample based</td>
<td>Yes</td>
</tr>
<tr>
<td>Frame based</td>
<td></td>
<td>No, produces an error</td>
</tr>
<tr>
<td>Sample based</td>
<td>Frame based</td>
<td>Yes</td>
</tr>
<tr>
<td>Frame based</td>
<td></td>
<td>Yes</td>
</tr>
</tbody>
</table>

For more information about these two processing modes, see “Sample- and Frame-Based Concepts” (DSP System Toolbox).

**Programmatic Use**

**Block Parameter:** InputProcessing  
*Type:* character vector  
*Values:* 'Columns as channels (frame based)' | 'Elements as channels (sample based)'  
*Default:* 'Elements as channels (sample based)'

**Use circular buffer for state — Circular buffer for storing state**

*off* (default) | *on*

Select to use a circular buffer for storing the state in simulation and code generation. Otherwise, an array buffer stores the state.

Using a circular buffer can improve execution speed when the delay length is large. For an array buffer, the number of copy operations increases as the delay length goes up. For a circular buffer, the number of copy operations is constant for increasing delay length.

If one of the following conditions is true, an array buffer always stores the state because a circular buffer does not improve execution speed.

- For sample-based signals, the delay length is 1.
- For frame-based signals, the delay length is no larger than the frame size.

**Programmatic Use**

**Block Parameter:** UseCircularBuffer  
*Type:* character vector  
*Values:* 'off' | 'on'
**Default:** 'off'

**Prevent direct feedthrough — Prevent direct feedthrough**

`off` (default) | `on`

Select to increase the delay length from zero to the lower limit for the **Input processing** mode.

- For sample-based signals, increase the minimum delay length to 1.
- For frame-based signals, increase the minimum delay length to the frame length.

Selecting this check box prevents direct feedthrough from the input port, `u`, to the output port. However, this check box cannot prevent direct feedthrough from the initial condition port, `x0`, to the output port.

**Dependency**

To enable this parameter, set **Delay length: Source** to **Input port**.

**Programmatic Use**

**Block Parameter:** `PreventDirectFeedthrough`

**Type:** character vector

**Values:** 'off' | 'on'

**Default:** 'off'

**Remove delay length check in generated code — Remove delay length out-of-range check**

`off` (default) | `on`

Select to remove code that checks for out-of-range delay length.

<table>
<thead>
<tr>
<th>Check Box</th>
<th>Result</th>
<th>When to Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selected</td>
<td>Generated code does not include conditional statements to check for out-of-range delay length.</td>
<td>For code efficiency</td>
</tr>
<tr>
<td>Cleared</td>
<td>Generated code includes conditional statements to check for out-of-range delay length.</td>
<td>For safety-critical applications</td>
</tr>
</tbody>
</table>
Dependency

To enable this parameter, set Delay length: Source to Input port.

Programmatic Use

Block Parameter: RemoveDelayLengthCheckInGeneratedCode
Type: character vector
Values: 'off' | 'on'
Default: 'off'

Diagnostic for delay length — Diagnostic checks for delay length
None (default) | Warning | Error

Specify whether to produce a warning or error when the input d is less than the lower limit or greater than the Delay length: Upper limit. The lower limit depends on the setting for Prevent direct feedthrough.

- If the check box is cleared, the lower limit is zero.
- If the check box is selected, the lower limit is 1 for sample-based signals and frame length for frame-based signals.

Options for the diagnostic include:

- None — Simulink software takes no action.
- Warning — Simulink software displays a warning and continues the simulation.
- Error — Simulink software terminates the simulation and displays an error.

Dependency

To enable this parameter, set Delay length: Source to Input port.

Programmatic Use

Block Parameter: DiagnosticForDelayLength
Type: character vector
Values: 'None' | 'Warning' | 'Error'
Default: 'None'

Show enable port — Create enable port
off (default) | on

Select to control execution of this block with an enable port. The block is considered enabled when the input to this port is nonzero, and is disabled when the input is 0. The value of the input is checked at the same time step as the block execution.
Programmatic Use
Block Parameter: ShowEnablePort
Type: character vector
Values: 'off' | 'on'
Default: 'off'

External reset — External state reset
None (default) | Rising | Falling | Either | Level | Level hold

Specify the trigger event to use to reset the states to the initial conditions.

<table>
<thead>
<tr>
<th>Reset Mode</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>No reset</td>
</tr>
<tr>
<td>Rising</td>
<td>Reset on a rising edge</td>
</tr>
<tr>
<td>Falling</td>
<td>Reset on a falling edge</td>
</tr>
<tr>
<td>Either</td>
<td>Reset on either a rising or falling edge</td>
</tr>
<tr>
<td>Level</td>
<td>Reset in either of these cases:</td>
</tr>
<tr>
<td></td>
<td>• When the reset signal is nonzero at the current time step</td>
</tr>
<tr>
<td></td>
<td>• When the reset signal value changes from nonzero at the previous time step to zero at the current time step</td>
</tr>
<tr>
<td>Level hold</td>
<td>Reset when the reset signal is nonzero at the current time step</td>
</tr>
</tbody>
</table>

Programmatic Use
Block Parameter: ExternalReset
Type: character vector
Values: 'None' | 'Rising' | 'Falling' | 'Either' | 'Level' | 'Level hold'
Default: 'None'

Sample time (-1 for inherited) — Discrete interval between sample time hits
-1 (default) | scalar

Specify the time interval between samples. To inherit the sample time, set this parameter to -1. This block supports discrete sample time, but not continuous sample time.
**Programmatic Use**

**Block Parameter:** SampleTime  
**Type:** character vector  
**Value:** real scalar  
**Default:** '-1'

**State Attributes**

**State name — Unique name for block state**

' ' (default) | alphanumeric string

Use this parameter to assign a unique name to the block state. The default is ' '. When this field is blank, no name is assigned. When using this parameter, remember these considerations:

- A valid identifier starts with an alphabetic or underscore character, followed by alphanumeric or underscore characters.
- The state name applies only to the selected block.

This parameter enables **State name must resolve to Simulink signal object** when you click **Apply**.

For more information, see “Apply Storage Classes to Individual Signal, State, and Parameter Data Elements” (Simulink Coder).

**Programmatic Use**

**Block Parameter:** StateName  
**Type:** character vector  
**Values:** unique name  
**Default:** ''

**State name must resolve to Simulink signal object — Require state name resolve to a signal object**

off (default) | on

Select this check box to require that the state name resolves to a Simulink signal object.

**Dependencies**

To enable this parameter, specify a value for **State name**. This parameter appears only if you set the model configuration parameter **Signal resolution** to a value other than None.
Selecting this check box disables **Code generation storage class**.

**Programmatic Use**

**Block Parameter:** StateMustResolveToSignalObject

**Type:** character vector

**Values:** 'off' | 'on'

**Default:** 'off'

**Signal object class — Custom storage class package name**

Simulink.Signal (default) | <StorageClass.PackageName>

Choose a custom storage class package by selecting a signal object class that the target package defines. For example, to apply custom storage classes from the built-in package mpt, select mpt.Signal. Unless you use an ERT-based code generation target with Embedded Coder, custom storage classes do not affect the generated code.

If the class that you want does not appear in the list, select Customize class lists. For instructions, see “Target Class Does Not Appear in List of Signal Object Classes” (Embedded Coder).

For information about storage classes, see “Apply Storage Classes to Individual Signal, State, and Parameter Data Elements” (Simulink Coder). For information about custom storage classes, see “Apply Built-In and Customized Storage Classes to Data Elements” (Embedded Coder).

**Programmatic Use**

**Block Parameter:** StateSignalObject

**Type:** character vector

**Values:** 'Simulink.Signal' | '<StorageClass.PackageName>'

**Default:** 'Simulink.Signal'

**Code generation storage class — State storage class for code generation**

Auto (default) | Model default | ExportedGlobal | ImportedExtern | ImportedExternPointer | BitField (Custom) | Model default | ExportToFile (Custom) | ImportFromFile (Custom) | FileScope (Custom) | AutoScope (Custom) | Struct (Custom) | GetSet (Custom) | Reusable (Custom)

Select state storage class for code generation.

- **Auto** is the appropriate storage class for states that you do not need to interface to external code.
- **StorageClass** applies the storage class or custom storage class that you select from the list. For information about storage classes, see “Apply Storage Classes to
Individual Signal, State, and Parameter Data Elements” (Simulink Coder). For information about custom storage classes, see “Apply Built-In and Customized Storage Classes to Data Elements” (Embedded Coder).

Use **Signal object class** to select custom storage classes from a package other than Simulink.

**Dependencies**

To enable this parameter, specify a value for **State name**.

**Programmatic Use**

**Block Parameter:** StateStorageClass

**Type:** character vector

**Values:** 'Auto' | 'Model default' | 'ExportedGlobal' | 'ImportedExtern' | 'ImportedExternPointer' | 'Custom' | ...

**Default:** 'Auto'

**Block Characteristics**

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>bus</th>
<th>double</th>
<th>enumerated</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Extended Capabilities**

**C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.
Generated code relies on `memcpy` or `memset` functions (`string.h`) under certain conditions.

**HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

<table>
<thead>
<tr>
<th>Block Parameter Setting</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set <strong>External reset</strong> to Level.</td>
<td>Generates a reset port in the HDL code.</td>
</tr>
<tr>
<td>Select <strong>Show enable port</strong>.</td>
<td>Generates an enable port in the HDL code.</td>
</tr>
<tr>
<td>For <strong>Initial condition</strong>, set <strong>Source</strong> to <strong>Dialog</strong> and enter the value.</td>
<td>Specifies an initial condition for the block.</td>
</tr>
<tr>
<td>Set <strong>Input processing</strong> to <strong>Columns as channels (frame based) or Elements as channels (sample based)</strong>.</td>
<td>Expects vector input data, where each element of the vector represents a sample in time.</td>
</tr>
</tbody>
</table>

**Additional Settings When Using State Control Block**

If you use a State Control block with the Delay block inside a subsystem in your Simulink model, use these additional settings.

<table>
<thead>
<tr>
<th>Block Parameter Setting</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set <strong>External reset</strong> to Level hold for <strong>Synchronous</strong> mode and Level for <strong>Classic</strong> mode of the State Control block.</td>
<td>Generates a reset port in the HDL code.</td>
</tr>
<tr>
<td>Set <strong>Delay length</strong> to zero for a Delay block with an external enable port.</td>
<td>Treated as a wire in only <strong>Synchronous</strong> mode of the State Control block.</td>
</tr>
<tr>
<td>Set <strong>Delay length</strong> to zero for a Delay block with an external reset port.</td>
<td>Treated as a wire in <strong>Synchronous</strong> and <strong>Classic</strong> modes of the State Control block.</td>
</tr>
</tbody>
</table>

For more information about the State Control block, see State Control.

**HDL Architecture**

This block has a single, default HDL architecture.
**HDL Block Properties**

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>InputPipeline</strong></td>
<td>Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “InputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td><strong>OutputPipeline</strong></td>
<td>Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “OutputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td><strong>ResetType</strong></td>
<td>Suppress reset logic generation. The default is default, which generates reset logic. See also “ResetType” (HDL Coder).</td>
</tr>
<tr>
<td><strong>UseRAM</strong></td>
<td>Map delays to RAM instead of registers. The default is off. See also “UseRAM” (HDL Coder).</td>
</tr>
</tbody>
</table>

**Complex Data Support**

This block supports code generation for complex signals.

**Restrictions**

For **Initial condition** and **Delay length, Source** set to **Input port** is not supported for HDL code generation.

**PLC Code Generation**

Generate Structured Text code using Simulink® PLC Coder™.

**Fixed-Point Conversion**

Design and simulate fixed-point systems using Fixed-Point Designer™.

**See Also**

Resettable Delay | Tapped Delay | Unit Delay | Variable Integer Delay

**Topics**

“Using Enabled Subsystems”
Introduced before R2006a
Demux

Extract and output elements of virtual vector signal

Library:  Simulink / Commonly Used Blocks
         Simulink / Signal Routing
         HDL Coder / Commonly Used Blocks
         HDL Coder / Signal Routing

Description

The Demux block extracts the components of an input vector signal and outputs separate signals. The output signal ports are ordered from top to bottom. See “Mux Signals” for information about creating and decomposing vectors.

Ports

Input

Port_1 — Accept nonbus vector signal to extract and output signals from
         real or complex values of any nonbus data type supported by Simulink

Vector input signal from which the Demux block selects scalar signals or smaller vectors.

Output

Port_1 — Output signals extracted from input vector signal
         nonbus signal with real or complex values of any data type supported by Simulink

Output signals extracted from the input vector. The output signal ports are ordered from top to bottom. See “Port Location After Rotating or Flipping” for a description of the port order for various block orientations.
Parameters

**Number of outputs — Number of outputs**

2 (default) | scalar | vector

Specify the number and, optionally, the dimensionality of each output port. If you do not specify the dimensionality of the outputs, the block determines the dimensionality of the outputs.

The value can be a scalar specifying the number of outputs or a vector whose elements specify the widths of the block output ports. The block determines the size of its outputs from the size of the input signal and the value of the **Number of outputs** parameter.

If you specify a scalar for the **Number of outputs** parameter and all of the output ports are connected, as you draw a new signal line close to output side of a Demux block, Simulink adds a port and updates the **Number of outputs** parameter.

For an input vector of width \( n \), here is what the block outputs.

<table>
<thead>
<tr>
<th>Parameter Value</th>
<th>Block outputs...</th>
<th>Examples and Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>( p = n )</td>
<td>( p ) scalar signals</td>
<td>If the input is a three-element vector and you specify three outputs, the block outputs three scalar signals.</td>
</tr>
<tr>
<td>( p &gt; n )</td>
<td>Error</td>
<td>This value is not supported.</td>
</tr>
<tr>
<td>( p &lt; n )</td>
<td>( p ) vector signals each having ( n/p ) elements</td>
<td>If the input is a six-element vector and you specify three outputs, the block outputs three two-element vectors.</td>
</tr>
<tr>
<td>( n \mod p = 0 )</td>
<td>( m ) vector signals each having ((n/p) + 1) elements and ( p - m ) signals having ( n/p ) elements</td>
<td>If the input is a five-element vector and you specify three outputs, the block outputs two two-element vector signals and one scalar signal.</td>
</tr>
<tr>
<td>Parameter Value</td>
<td>Block outputs...</td>
<td>Examples and Comments</td>
</tr>
<tr>
<td>-----------------</td>
<td>------------------</td>
<td>----------------------</td>
</tr>
<tr>
<td>([p_1 \ p_2 \ldots p_m])</td>
<td>(m) vector signals having widths (p_1, p_2, \ldots p_m)</td>
<td>If the input is a five-element vector and you specify ([3, 2]) as the output, the block outputs three of the input elements on one port and the other two elements on the other port.</td>
</tr>
<tr>
<td>(p_1+p_2+\ldots+p_m=n)</td>
<td>(p_i &gt; 0)</td>
<td>An array that has one or more of (m) elements with a value of (-1), which specifies that Simulink infers the size for the element. For example, suppose that you have a four-element array with a total width of 14 and you specify the parameter to be ([p_1 p_2 \ -1 \ p_4]). The value for the third element (the (-1) element) is (14 - (p_1 + p_2 + p_4)).</td>
</tr>
<tr>
<td>([p_1 \ p_2 \ldots p_m])</td>
<td>(p_1+p_2+\ldots+p_m!=n)</td>
<td>This value is not supported</td>
</tr>
<tr>
<td>(p_i = \theta)</td>
<td>Error</td>
<td></td>
</tr>
</tbody>
</table>

If you specify the number of outputs that is smaller than the number of input elements, the block distributes the elements as evenly as possible over the outputs. For examples, see “Extract Vector Elements and Distribute Evenly Across Outputs” on page 14-188 and “Extract Vector Elements Using the Demux Block” on page 14-189.

**Programmatic Use**

**Block Parameter:** Outputs  
**Type:** scalar or vector  
**Values:** character vector  
**Default:** \{"2\} or vector

**Display option — Displayed block icon**  
bar (default) | none
By default, the block icon is a solid bar of the block foreground color. To display the icon as a box containing the block type name, select none.

**Programmatic Use**

**Block Parameter:** Display option  
**Type:** character vector  
**Values:** 'bar' | 'none'  
**Default:** 'bar'

---

**Block Characteristics**

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>double</th>
<th>enumerated</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Direct Feedthrough</th>
<th>yes</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Multidimensional Signals</th>
<th>no</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Variable-Size Signals</th>
<th>no</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Zero-Crossing Detection</th>
<th>no</th>
</tr>
</thead>
</table>

---

**Extended Capabilities**

**C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

Actual data type or capability support depends on block implementation.

**HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.
HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

**HDL Architecture**

This block has a single, default HDL architecture.

**HDL Block Properties**

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConstrainedOutputPipeline</td>
<td>Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. For more details, see “ConstrainedOutputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td>InputPipeline</td>
<td>Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “InputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td>OutputPipeline</td>
<td>Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “OutputPipeline” (HDL Coder).</td>
</tr>
</tbody>
</table>

**Complex Data Support**

This block supports code generation for complex signals.

**PLC Code Generation**

Generate Structured Text code using Simulink® PLC Coder™.

**Fixed-Point Conversion**

Design and simulate fixed-point systems using Fixed-Point Designer™.

Actual data type or capability support depends on block implementation.

**See Also**

Bus Creator | Bus to Vector | Mux
Topics
“Virtual Signals”
“Simplify Bus Interfaces”
“Composite Signals”
feedbacksystem

Introduced before R2006a
Derivative

Output time derivative of input

Library: Simulink / Continuous

Description

The Derivative block approximates the derivative of the input signal \( u \) with respect to the simulation time \( t \). You obtain the approximation of \( \frac{du}{dt} \) by computing a numerical difference \( \Delta u / \Delta t \), where \( \Delta u \) is the change in input value and \( \Delta t \) is the change in time since the previous simulation (major) time step.

This block accepts one input and generates one output. The initial output for the block is zero.

The precise relationship between the input and output of this block is:

\[
y(t) = \frac{\Delta u}{\Delta t} = \frac{u(t) - u(T_{\text{previous}})}{t - T_{\text{previous}}} \quad t > T_{\text{previous}},
\]

where \( t \) is the current simulation time and \( T_{\text{previous}} \) is the time of the last output time of the simulation. The latter is the same as the time of the last major time step.

The Derivative block output might be sensitive to the dynamics of the entire model. The accuracy of the output signal depends on the size of the time steps taken in the simulation. Smaller steps allow for a smoother and more accurate output curve from this block. However, unlike with blocks that have continuous states, the solver does not take smaller steps when the input to this block changes rapidly. Depending on the dynamics of the driving signal and model, the output signal of this block might contain unexpected fluctuations. These fluctuations are primarily due to the driving signal output and solver step size.

Because of these sensitivities, structure your models to use integrators (such as Integrator blocks) instead of Derivative blocks. Integrator blocks have states that allow
solvers to adjust the step size and improve simulation accuracy. See “Circuit Model” for an example of choosing the best-form mathematical model to avoid using Derivative blocks in your models.

If you must use the Derivative block with a variable step solver, set the solver maximum step size to a value such that the Derivative block can generate answers with adequate accuracy. To determine this value, you might need to repeatedly run the simulation using different solver settings.

If the input to this block is a discrete signal, the continuous derivative of the input exhibits an impulse when the value of the input changes. Otherwise, it is 0. Alternatively, you can define the discrete derivative of a discrete signal using the difference of the last two values of the signal:

\[ y(k) = \frac{1}{\Delta t} (u(k) - u(k-1)) \]

Taking the z-transform of this equation results in:

\[ \frac{Y(z)}{u(z)} = \frac{1 - z^{-1}}{\Delta t} = \frac{z - 1}{\Delta t \cdot z} . \]

The Discrete Derivative block models this behavior. Use this block instead of the Derivative block to approximate the discrete-time derivative of a discrete signal.

**Ports**

**Input**

**Port_1 — Input signal**
real scalar or vector

Signal to be differentiated, specified as a real scalar or vector.

Data Types: double
Output

**Port_1 — Time derivative of input signal**
real scalar or vector

Time derivative of input signal, specified as a real scalar or vector. The input signal is differentiated with respect to time as:

\[ y(t) = \frac{\Delta u}{\Delta t} = \frac{u(t) - u(T_{\text{previous}})}{t - T_{\text{previous}}} \mid t > T_{\text{previous}}, \]

where \( t \) is the current simulation time and \( T_{\text{previous}} \) is the time of the last output time of the simulation. The latter is the same as the time of the last major time step.

Data Types: double

Parameters

**Coefficient c in the transfer function approximation \( s/(c*s + 1) \) used for linearization — Specify the time constant c to approximate the linearization of your system**
inf (default)

The exact linearization of the Derivative block is difficult because the dynamic equation for the block is \( y = \dot{u} \), which you cannot represent as a state-space system. However, you can approximate the linearization by adding a pole to the Derivative block to create a transfer function \( s/(c * s + 1) \). The addition of a pole filters the signal before differentiating it, which removes the effect of noise.

The default value \( \text{inf} \) corresponds to a linearization of \( 0 \).

**Tips**

- As a best practice, change the value of \( c \) to \( \frac{1}{f_b} \), where \( f_b \) is the break frequency of the filter.
- The parameter must be a finite positive value.

**Programmatic Use**
**Block Parameter:** CoefficientInTFapproximation
**Type**: character vector, string  
**Values**: 'inf'  
**Default**: 'inf'

### Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>double</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>no</td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
</tr>
</tbody>
</table>

### Extended Capabilities

**C/C++ Code Generation**  
Generate C and C++ code using Simulink® Coder™.

Consider using the Model Discretizer to map the continuous blocks into discrete equivalents that support code generation. To access the Model Discretizer, in the Apps tab, under Control Systems, click Model Discretizer.

Not recommended for production code.

### See Also

Discrete Derivative
Topics
“Improved Linearization with Transfer Fcn Blocks” on page 14-39

Introduced before R2006a
Descriptor State-Space

Model linear implicit systems

Library: Simulink / Continuous

Description

The Descriptor State-Space block allows you to model linear implicit systems that can be expressed in the form $E \dot{x} = Ax + Bu$ where $E$ is the mass matrix of the system. When $E$ is nonsingular and therefore invertible, the system can be written in its explicit form $\dot{x} = E^{-1}Ax + E^{-1}Bu$ and modeled using the State-Space block.

When the mass matrix $E$ is singular, one or more derivatives of the dependent variables of the system are not present in the equations. These variables are called algebraic variables. Differential equations that contain such algebraic variables are called differential algebraic equations. Their state space representation is of the form

$E \dot{x} = Ax + Bu$
$y = Cx + Du$

where the variables have the following meanings:

- $x$ is the state vector
- $u$ is the input vector
- $y$ is the output vector
Ports

Input

Input 1 — Input signal
scalar | vector

Real-valued input vector of type double whose width is the number of columns in the B and D matrices.

Data Types: double

Output

Output 1 — Output vector
scalar | vector

Real-valued input vector of type double whose width is the number of rows in the C and D matrices.

Data Types: double

Parameters

E — Mass matrix
1 (default) | scalar | matrix

Specify the mass matrix E as a real-valued n-by-n matrix, where n is the number of states in the system. E must be the same size as A. E can be singular or non-singular.

Programmatic Use
Block Parameter: E
Type: character vector, string
Values: scalar | matrix
Default: '1'

A — Matrix coefficient, A
1 (default) | scalar | matrix
Specify the matrix coefficient $A$ as a real-valued $n$-by-$n$ matrix, where $n$ is the number of states in the system. $A$ must be the same size as $E$.

**Programmatic Use**

**Block Parameter:** $A$

- **Type:** character vector, string
- **Values:** scalar | matrix
- **Default:** '1'

$B$ — Matrix coefficient, $B$

1 (default) | scalar | vector | matrix

Specify the matrix coefficient $B$ as a real-valued $n$-by-$m$ matrix, where $n$ is the number of states in the system and $m$ is the number of inputs.

**Programmatic Use**

**Block Parameter:** $B$

- **Type:** character vector, string
- **Values:** scalar | vector | matrix
- **Default:** '1'

$C$ — Matrix coefficient, $C$

1 (default) | scalar | vector | matrix

Specify the matrix coefficient $C$ as a real-valued $r$-by-$n$ matrix, where $n$ is the number of states in the system and $r$ is the number of outputs.

**Programmatic Use**

**Block Parameter:** $C$

- **Type:** character vector, string
- **Values:** scalar | vector | matrix
- **Default:** '1'

$D$ — Matrix coefficient, $D$

1 (default) | scalar | vector | matrix

Specify the matrix coefficient $D$ as a real-valued $r$-by-$m$ matrix, where $r$ is the number of outputs of the system and $m$ is the number of inputs to the system.

**Programmatic Use**

**Block Parameter:** $D$

- **Type:** character vector, string
- **Values:** scalar | vector | matrix
**Default:** '1'

**Initial condition — Initial condition of states**

0 (default) | scalar | vector | matrix

Specify initial condition of the block states. The minimum and maximum values are bound by the **Output minimum** and **Output maximum** block parameters.

**Tip** Simulink software does not allow the initial condition of this block to be `inf` or `NaN`.

**Dependencies**

To enable this parameter, set the **Initial condition source** to **internal**.

**Programmatic Use**

**Block Parameter:** InitialCondition

**Type:** character vector, string  
**Values:** scalar | vector | matrix  
**Default:** '0'

**Direct feedthrough — Set output signal dependency on input**

true (default) | false

Specify whether the output of the block has a direct dependency on the input signal. Use this parameter for systems having more than 500 continuous states in order to speed up simulation. For systems with 500 continuous states or less, Simulink automatically determines this setting.

**Programmatic Use**

**Block Parameter:** DirectFeedthrough

**Type:** character vector, string  
**Values:** 'True' | 'False'  
**Default:** 'True'

**Absolute tolerance — Absolute tolerance for computing block states**

auto (default) | scalar | vector

Absolute tolerance for computing block states, specified as a positive, real-valued, scalar or vector. To inherit the absolute tolerance from the Configuration Parameters, specify `auto` or `-1`. 
• If you enter a real scalar, then that value overrides the absolute tolerance in the Configuration Parameters dialog box for computing all block states.

• If you enter a real vector, then the dimension of that vector must match the dimension of the continuous states in the block. These values override the absolute tolerance in the Configuration Parameters dialog box.

• If you enter auto or -1, then Simulink uses the absolute tolerance value in the Configuration Parameters dialog box (see “Solver Pane”) to compute block states.

Programmatic Use
Block Parameter: AbsoluteTolerance
Type: character vector, string
Values: 'auto' | '-1' | any positive real-valued scalar or vector
Default: 'auto'

State Name (e.g., 'position') — Assign unique name to each state
'default' | 'position' | {'a', 'b', 'c'} | a | ...

Assign a unique name to each state. If this field is blank (' '), no name assignment occurs.

• To assign a name to a single state, enter the name between quotes, for example, 'position'.

• To assign names to multiple states, enter a comma-delimited list surrounded by braces, for example, {'a', 'b', 'c'}. Each name must be unique.

• To assign state names with a variable in the MATLAB workspace, enter the variable without quotes. A variable can be a character vector, string, cell array, or structure.

Limitations
• The state names apply only to the selected block.

• The number of states must divide evenly among the number of state names.

• You can specify fewer names than states, but you cannot specify more names than states.

For example, you can specify two names in a system with four states. The first name applies to the first two states and the second name to the last two states.

Programmatic Use
Block Parameter: ContinuousStateAttributes
Type: character vector, string
Values: ' ' | user-defined
Default: ' ' 

Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>double</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>yes</td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>no</td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
</tr>
</tbody>
</table>

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

1 Consider using the Model Discretizer to map these continuous blocks into discrete equivalents that support code generation. To access the Model Discretizer, in the Apps tab, under Control Systems, click Model Discretizer.

2 Not recommended for production code.

See Also

Blocks
Algebraic Constraint | State-Space

Functions
dss
Topics
“Solve Differential Algebraic Equations (DAEs)” (MATLAB)
“Model Differential Algebraic Equations”

External Websites
https://www.mathworks.com/matlabcentral/fileexchange/7481-manuscript-of-solving-index-1-daes-in-matlab-and-simulink-

Introduced in R2018b
Detect Change

Detect change in signal value

Library: Simulink / Logic and Bit Operations
         HDL Coder / Logic and Bit Operations

Description

The Detect Change block determines if an input signal does not equal its previous value. The initial condition determines the initial value of the previous input U/z.

Ports

Input

Port_1 — Input signal
    signal value

Input signal, specified as a scalar, vector, or matrix.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 |
             Boolean | fixed point | enumerated | built-in integer | floating point

Output

Port_1 — Output signal
    0 | 1

Output signal, true (equal to 1) when the input signal does not equal its previous value; false (equal to 0) when the input signal equals its previous value.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 |
             Boolean | fixed point | enumerated | bus
Parameters

Initial condition — Initial condition for the previous input
0 (default) | scalar | vector

Set the initial condition for the previous input U/z.

Programmatic Use
Block Parameter: vinit
Type: character vector
Values: scalar | vector
Default: '0'

Input processing — Specify sample- or frame-based processing
Elements as channels (sample based) (default) | Columns as channels (frame based)

Specify whether the block performs sample- or frame-based processing:

- Columns as channels (frame based) — Treat each column of the input as a separate channel (frame-based processing).

Note Frame-based processing requires a DSP System Toolbox license.

For more information, see “Sample- and Frame-Based Concepts” (DSP System Toolbox).

- Elements as channels (sample based) — Treat each element of the input as a separate channel (sample-based processing).

Use Input processing to specify whether the block performs sample- or frame-based processing. The block accepts frame-based signals for the input u. All other input signals must be sample-based.

<table>
<thead>
<tr>
<th>Input Signal u</th>
<th>Input Processing Mode</th>
<th>Block Works?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sample based</td>
<td>Sample based</td>
<td>Yes</td>
</tr>
<tr>
<td>Frame based</td>
<td></td>
<td>No, produces an error</td>
</tr>
<tr>
<td>Sample based</td>
<td>Frame based</td>
<td>Yes</td>
</tr>
<tr>
<td>Frame based</td>
<td></td>
<td>Yes</td>
</tr>
</tbody>
</table>
For more information about these two processing modes, see “Sample- and Frame-Based Concepts” (DSP System Toolbox).

**Programmatic Use**

**Block Parameter:** InputProcessing  
**Type:** character vector  
**Values:**  
- 'Columns as channels (frame based)'  
- 'Elements as channels (sample based)'  
**Default:** 'Elements as channels (sample based)'

**Output data type — Data type of the output**

| boolean (default) | uint8 |

Set the output data type to boolean or uint8.

**Programmatic Use**

**Block Parameter:** OutDataTypeStr  
**Type:** character vector  
**Values:**  
- 'boolean'  
- 'uint8'  
**Default:** 'boolean'

**Block Characteristics**

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>double</th>
<th>enumerated</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

Generated code relies on `memcpy` or `memset` functions (`string.h`) under certain conditions.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture
This block has a single, default HDL architecture.

HDL Block Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConstrainedOutputPipeline</td>
<td>Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. For more details, see “ConstrainedOutputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td>InputPipeline</td>
<td>Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “InputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td>OutputPipeline</td>
<td>Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “OutputPipeline” (HDL Coder).</td>
</tr>
</tbody>
</table>

PLC Code Generation
Generate Structured Text code using Simulink® PLC Coder™.
Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

See Also
Detect Decrease | Detect Fall Negative | Detect Fall Nonpositive | Detect Increase | Detect Rise Nonnegative | Detect Rise Positive

Introduced before R2006a
Detect Decrease

Detect decrease in signal value

Library:

Simulink / Logic and Bit Operations
HDL Coder / Logic and Bit Operations

Description

The Detect Decrease block determines if an input is strictly less than its previous value.

Ports

Input

Port_1 — Input signal
scalar | vector | matrix

Input signal, specified as a scalar, vector, or matrix.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 |
Boolean | fixed point | enumerated

Output

Port_1 — Output signal
0 | 1

Output signal, true (equal to 1) when the input signal is less than its previous value; false (equal to 0) when the input signal is greater than or equal to its previous value.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 |
Boolean | fixed point | enumerated
Parameters

Initial condition — Initial condition for the previous input
0 (default) | scalar | vector

Set the initial condition for the previous input U/z.

Programmatic Use
Block Parameter: vinit
Type: character vector
Values: scalar | vector
Default: '0'

Input processing — Specify sample- or frame-based processing
Elements as channels (sample based) (default) | Columns as channels (frame based)

Specify whether the block performs sample- or frame-based processing:

- Columns as channels (frame based) — Treat each column of the input as a separate channel (frame-based processing).

Note Frame-based processing requires a DSP System Toolbox license.

For more information, see “Sample- and Frame-Based Concepts” (DSP System Toolbox).

- Elements as channels (sample based) — Treat each element of the input as a separate channel (sample-based processing).

Use Input processing to specify whether the block performs sample- or frame-based processing. The block accepts frame-based signals for the input u. All other input signals must be sample-based.

<table>
<thead>
<tr>
<th>Input Signal u</th>
<th>Input Processing Mode</th>
<th>Block Works?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sample based</td>
<td>Sample based</td>
<td>Yes</td>
</tr>
<tr>
<td>Frame based</td>
<td></td>
<td>No, produces an error</td>
</tr>
<tr>
<td>Sample based</td>
<td>Frame based</td>
<td>Yes</td>
</tr>
<tr>
<td>Frame based</td>
<td></td>
<td>Yes</td>
</tr>
</tbody>
</table>
For more information about these two processing modes, see “Sample- and Frame-Based Concepts” (DSP System Toolbox).

**Programmatic Use**
**Block Parameter:** InputProcessing
**Type:** character vector
**Values:** 'Columns as channels (frame based)' | 'Elements as channels (sample based)'
**Default:** 'Elements as channels (sample based)'

**Output data type — Data type of the output**
boolean (default) | uint8

Set the output data type to boolean or uint8.

**Programmatic Use**
**Block Parameter:** OutDataTypeStr
**Type:** character vector
**Values:** 'boolean' | 'uint8'
**Default:** 'boolean'

**Block Characteristics**

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>double</th>
<th>enumerated</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Extended Capabilities

#### C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

Generated code relies on `memcpy` or `memset` functions (`string.h`) under certain conditions.

#### HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

**HDL Architecture**

This block has a single, default HDL architecture.

**HDL Block Properties**

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConstrainedOutputPipeline</td>
<td>Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. For more details, see “ConstrainedOutputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td>InputPipeline</td>
<td>Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “InputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td>OutputPipeline</td>
<td>Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “OutputPipeline” (HDL Coder).</td>
</tr>
</tbody>
</table>

#### PLC Code Generation
Generate Structured Text code using Simulink® PLC Coder™.
Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

See Also
Detect Change | Detect Fall Negative | Detect Fall Nonpositive | Detect Increase | Detect Rise Nonnegative | Detect Rise Positive

Introduced before R2006a
Detect Fall Negative

Detect falling edge when signal value decreases to strictly negative value, and its previous value was nonnegative

**Library:** Simulink / Logic and Bit Operations

**Description**

The Detect Fall Negative block determines if the input is less than zero, and its previous value is greater than or equal to zero.

**Ports**

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

**Input**

Port 1 — Input signal

signal value

Input signal, specified as a scalar, vector, or matrix.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated

**Output**

Port 1 — Output signal

0 | 1
Output signal, true (equal to 1) when the input signal is less than zero, and its previous value was greater than or equal to zero; false (equal to 0) when the input signal is greater than or equal to zero, or if the input signal is negative, its previous value was also negative.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus

Parameters

**Initial condition** — Initial condition for the previous input

0 (default) | scalar | vector

Set the initial condition of the Boolean expression \( \frac{U}{z} < 0 \).

**Programmatic Use**

**Block Parameter:** vinit

**Type:** character vector

**Values:** scalar | vector

**Default:** '0'

**Input processing** — Specify sample- or frame-based processing

Elements as channels (sample based) (default) | Columns as channels (frame based)

Specify whether the block performs sample- or frame-based processing:

- **Columns as channels (frame based)** — Treat each column of the input as a separate channel (frame-based processing).

**Note** Frame-based processing requires a DSP System Toolbox license.

For more information, see “Sample- and Frame-Based Concepts” (DSP System Toolbox).

- **Elements as channels (sample based)** — Treat each element of the input as a separate channel (sample-based processing).

Use **Input processing** to specify whether the block performs sample- or frame-based processing. The block accepts frame-based signals for the input \( u \). All other input signals must be sample-based.
### Input Signal \( u \) vs. Input Processing Mode vs. Block Works?

<table>
<thead>
<tr>
<th>Input Signal ( u )</th>
<th>Input Processing Mode</th>
<th>Block Works?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sample based</td>
<td>Sample based</td>
<td>Yes</td>
</tr>
<tr>
<td>Frame based</td>
<td></td>
<td>No, produces an error</td>
</tr>
<tr>
<td>Sample based</td>
<td>Frame based</td>
<td>Yes</td>
</tr>
<tr>
<td>Frame based</td>
<td></td>
<td>Yes</td>
</tr>
</tbody>
</table>

For more information about these two processing modes, see “Sample- and Frame-Based Concepts” (DSP System Toolbox).

**Programmatic Use**

**Block Parameter:** InputProcessing

**Type:** character vector

**Values:** 'Columns as channels (frame based)' | 'Elements as channels (sample based)'

**Default:** 'Elements as channels (sample based)'

**Output data type — Data type of the output**

boolean (default) | uint8

Set the output data type to boolean or uint8.

**Programmatic Use**

**Block Parameter:** OutDataTypeStr

**Type:** character vector

**Values:** 'boolean' | 'uint8'

**Default:** 'boolean'

### Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>double</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

Generated code relies on memcpy or memset functions (string.h) under certain conditions.

PLC Code Generation
Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

See Also
Detect Change | Detect Decrease | Detect Fall Nonpositive | Detect Increase | Detect Rise Nonnegative | Detect Rise Positive

Introduced before R2006a
Detect Fall Nonpositive

Detect falling edge when signal value decreases to nonpositive value, and its previous value was strictly positive

**Library:** Simulink / Logic and Bit Operations

**Description**

The Detect Fall Nonpositive block determines if the input is less than or equal to zero, and its previous value was greater than zero.

- The output is true (equal to 1) when the input signal is less than or equal to zero, and its previous value was greater than zero.
- The output is false (equal to 0) when the input signal is greater than zero, or if it is nonpositive, its previous value was also nonpositive.

**Ports**

**Input**

**Port 1 — Input signal**

scalar | vector | matrix

Input signal, specified as a scalar, vector, or matrix.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point
**Output**

**Port_1 — Output signal**
scalar | vector | matrix

Output signal that detects a falling edge, specified as a scalar, vector, or matrix.

- The output is true (equal to 1) when the input signal is less than or equal to zero, and its previous value was greater than zero.
- The output is false (equal to 0) when the input signal is greater than zero, or if it is nonpositive, its previous value was also nonpositive.

Data Types: uint8 | Boolean

**Parameters**

**Initial condition — Initial condition of Boolean expression U/z <= 0**
0 (default) | scalar | vector | matrix

Set the initial condition of the Boolean expression U/z <= 0.

**Programmatic Use**
**Block Parameter:** vinit
**Type:** character vector
**Values:** scalar | vector | matrix
**Default:** '0'

**Input processing — Specify sample- or frame-based processing**
Elements as channels (sample based) (default) | Columns as channels (frame based)

Specify whether the block performs sample- or frame-based processing:

- **Columns as channels (frame based)** — Treat each column of the input as a separate channel (frame-based processing).

**Note** Frame-based processing requires a DSP System Toolbox license.

For more information, see “Sample- and Frame-Based Concepts” (DSP System Toolbox).
• **Elements as channels (sample based)** — Treat each element of the input as a separate channel (sample-based processing).

Use **Input processing** to specify whether the block performs sample- or frame-based processing. The block accepts frame-based signals for the input \( u \). All other input signals must be sample-based.

<table>
<thead>
<tr>
<th>Input Signal ( u )</th>
<th>Input Processing Mode</th>
<th>Block Works?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sample based</td>
<td>Sample based</td>
<td>Yes</td>
</tr>
<tr>
<td>Frame based</td>
<td></td>
<td>No, produces an error</td>
</tr>
<tr>
<td>Sample based</td>
<td>Frame based</td>
<td>Yes</td>
</tr>
<tr>
<td>Frame based</td>
<td></td>
<td>Yes</td>
</tr>
</tbody>
</table>

For more information about these two processing modes, see “Sample- and Frame-Based Concepts” (DSP System Toolbox).

**Programmatic Use**

**Block Parameter:** `InputProcessing`

*Type:* character vector  
*Values:* `'Columns as channels (frame based)'` | `'Elements as channels (sample based)'`

*Default:* `'Elements as channels (sample based)'`

**Output data type** — **Output data type**

`boolean` (default) | `uint8`

Specify the output data type as `boolean` or `uint8`.

**Programmatic Use**

**Block Parameter:** `OutDataTypeStr`

*Type:* character vector  
*Values:* `'boolean'` | `'uint8'`

*Default:* `'boolean'`

**Block Characteristics**

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>double</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
</table>
Direct Feedthrough | yes
---|---
Multidimensional Signals | yes
Variable-Size Signals | yes
Zero-Crossing Detection | no

**Extended Capabilities**

**C/C++ Code Generation**
Generate C and C++ code using Simulink® Coder™.

Generated code relies on `memcpy` or `memset` functions (string.h) under certain conditions.

**PLC Code Generation**
Generate Structured Text code using Simulink® PLC Coder™.

**Fixed-Point Conversion**
Design and simulate fixed-point systems using Fixed-Point Designer™.

**See Also**
Detect Change | Detect Decrease | Detect Fall Negative | Detect Increase | Detect Rise Nonnegative | Detect Rise Positive

Introduced before R2006a
Detect Increase

Detect increase in signal value

**Library:**
- Simulink / Logic and Bit Operations
- HDL Coder / Logic and Bit Operations

**Description**

The Detect Increase block determines if an input is strictly greater than its previous value.

- The output is true (equal to 1) when the input signal is greater than its previous value.
- The output is false (equal to 0) when the input signal is less than or equal to its previous value.

**Ports**

**Input**

**Port_1 — Input signal**

<table>
<thead>
<tr>
<th>scalar</th>
<th>vector</th>
<th>matrix</th>
</tr>
</thead>
</table>

Input signal, specified as a scalar, vector, or matrix.

**Data Types:**
- single
- double
- int8
- int16
- int32
- uint8
- uint16
- uint32
- Boolean
- fixed point
- enumerated

**Output**

**Port_1 — Output signal**

<table>
<thead>
<tr>
<th>scalar</th>
<th>vector</th>
<th>matrix</th>
</tr>
</thead>
</table>


Output signal, detecting an increase in signal value, specified as a scalar, vector, or matrix.

- The output is true (equal to 1) when the input signal is greater than its previous value.
- The output is false (equal to 0) when the input signal is less than or equal to its previous value.

Data Types: `uint8` | `Boolean`

### Parameters

**Initial condition — Initial condition of previous input**

\[ 0.0 \text{ (default)} | \text{scalar} | \text{vector} | \text{matrix} \]

Set the initial condition for the previous input \( U/z \).

**Programmatic Use**

**Block Parameter:** `vinit`

**Type:** character vector

**Values:** scalar | vector | matrix

**Default:** `'0.0'`  

**Input processing — Specify sample- or frame-based processing**

Elements as channels (sample based) (default) | Columns as channels (frame based)

Specify whether the block performs sample- or frame-based processing:

- **Columns as channels (frame based)** — Treat each column of the input as a separate channel (frame-based processing).

**Note** Frame-based processing requires a DSP System Toolbox license.

For more information, see “Sample- and Frame-Based Concepts” (DSP System Toolbox).

- **Elements as channels (sample based)** — Treat each element of the input as a separate channel (sample-based processing).
Use **Input processing** to specify whether the block performs sample- or frame-based processing. The block accepts frame-based signals for the input \( u \). All other input signals must be sample-based.

<table>
<thead>
<tr>
<th>Input Signal ( u )</th>
<th>Input Processing Mode</th>
<th>Block Works?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sample based</td>
<td>Sample based</td>
<td>Yes</td>
</tr>
<tr>
<td>Frame based</td>
<td></td>
<td>No, produces an error</td>
</tr>
<tr>
<td>Sample based</td>
<td>Frame based</td>
<td>Yes</td>
</tr>
<tr>
<td>Frame based</td>
<td></td>
<td>Yes</td>
</tr>
</tbody>
</table>

For more information about these two processing modes, see “Sample- and Frame-Based Concepts” (DSP System Toolbox).

**Programmatic Use**

**Block Parameter:** `InputProcessing`  
**Type:** character vector  
**Values:** 'Columns as channels (frame based)' | 'Elements as channels (sample based)'  
**Default:** 'Elements as channels (sample based)'

**Output data type — Output data type**

`boolean` (default) | `uint8`

Specify the output data type as `boolean` or `uint8`.

**Programmatic Use**

**Block Parameter:** `OutDataTypeStr`  
**Type:** character vector  
**Values:** 'boolean' | 'uint8'  
**Default:** 'boolean'

**Block Characteristics**

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>double</th>
<th>enumerated</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-------------------------</td>
<td>-----</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Extended Capabilities

#### C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

Generated code relies on `memcpy` or `memset` functions (`string.h`) under certain conditions.

#### HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

#### HDL Architecture
This block has a single, default HDL architecture.

#### HDL Block Properties

| ConstrainedOutput Pipeline | Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. For more details, see “ConstrainedOutputPipeline” (HDL Coder). |
**InputPipeline**
Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “InputPipeline” (HDL Coder).

**OutputPipeline**
Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “OutputPipeline” (HDL Coder).

**PLC Code Generation**
Generate Structured Text code using Simulink® PLC Coder™.

**Fixed-Point Conversion**
Design and simulate fixed-point systems using Fixed-Point Designer™.

**See Also**
Detect Change | Detect Decrease | Detect Fall Negative | Detect Fall Nonpositive | Detect Rise Nonnegative | Detect Rise Positive

Introduced before R2006a
Detect Rise Nonnegative

Detect rising edge when signal value increases to nonnegative value, and its previous value was strictly negative

**Library:** Simulink / Logic and Bit Operations

### Description

The Detect Rise Nonnegative block determining if the input is greater than or equal to zero, and its previous value was less than zero.

- The output is true (equal to 1) when the input signal is greater than or equal to zero, and its previous value was less than zero.
- The output is false (equal to 0) when the input signal is less than zero, or if the input signal is nonnegative, its previous value was also nonnegative.

### Ports

**Input**

**Port_1 — Input signal**

Scalar | Vector | Matrix

Input signal, specified as a scalar, vector, or matrix.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point

1-417
Output

Port_1 — Output signal
scalar | vector | matrix

Output signal that indicates a rising edge whenever the signal value increases to a nonnegative value, and its previous value was strictly negative. The output can be a scalar, vector, or matrix.

• The output is true (equal to 1) when the input signal is greater than or equal to zero, and its previous value was less than zero.
• The output is false (equal to 0) when the input signal is less than zero, or if the input signal is nonnegative, its previous value was also nonnegative.

Data Types: uint8 | Boolean

Parameters

Initial condition — Initial condition of Boolean expression U/z >= 0
0 (default) | scalar | vector | matrix

Set the initial condition of the Boolean expression U/z >= 0.

Programmatic Use
Block Parameter: vinit
Type: character vector
Values: scalar | vector | matrix
Default: '0'

Input processing — Specify sample- or frame-based processing
Elements as channels (sample based) (default) | Columns as channels (frame based)

Specify whether the block performs sample- or frame-based processing:

• Columns as channels (frame based) — Treat each column of the input as a separate channel (frame-based processing).

Note Frame-based processing requires a DSP System Toolbox license.
For more information, see “Sample- and Frame-Based Concepts” (DSP System Toolbox).

- **Elements as channels (sample based)** — Treat each element of the input as a separate channel (sample-based processing).

Use **Input processing** to specify whether the block performs sample- or frame-based processing. The block accepts frame-based signals for the input \( u \). All other input signals must be sample-based.

<table>
<thead>
<tr>
<th>Input Signal ( u )</th>
<th>Input Processing Mode</th>
<th>Block Works?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sample based</td>
<td>Sample based</td>
<td>Yes</td>
</tr>
<tr>
<td>Frame based</td>
<td></td>
<td>No, produces an error</td>
</tr>
<tr>
<td>Sample based</td>
<td>Frame based</td>
<td>Yes</td>
</tr>
<tr>
<td>Frame based</td>
<td></td>
<td>Yes</td>
</tr>
</tbody>
</table>

For more information about these two processing modes, see “Sample- and Frame-Based Concepts” (DSP System Toolbox).

**Programmatic Use**

**Block Parameter:** InputProcessing

Type: character vector

Values: 'Columns as channels (frame based)' | 'Elements as channels (sample based)'

Default: 'Elements as channels (sample based)'

**Output data type** — **Output data type**

boolean (default) | uint8

Specify the output data type as boolean or uint8.

**Programmatic Use**

**Block Parameter:** OutDataTypeStr

Type: character vector

Values: 'boolean' | 'uint8'

Default: 'boolean'
Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>double</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

Generated code relies on memcpy or memset functions (string.h) under certain conditions.

PLC Code Generation
Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

See Also
Detect Change | Detect Decrease | Detect Fall Negative | Detect Fall Nonpositive | Detect Increase | Detect Rise Positive
Introduced before R2006a
Detect Rise Positive

Detect rising edge when signal value increases to strictly positive value, and its previous value was nonpositive

**Library:** Simulink / Logic and Bit Operations

**Description**

The Detect Rise Positive block detects a rising edge by determining if the input is strictly positive, and its previous value was nonpositive.

- The output is true (equal to 1) when the input signal is greater than zero, and the previous value was less than or equal to zero.
- The output is false (equal to 0) when the input is negative or zero, or if the input is positive, the previous value was also positive.

**Ports**

**Input**

**Port 1 — Input signal**

scalar | vector | matrix

Input signal, specified as a scalar, vector, or matrix.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point
**Output**

**Port_1 — Output signal**
scalar | vector | matrix

Output signal that detects a rising edge whenever the input is strictly positive, and its previous value was nonpositive. The output can be a scalar, vector, or matrix.

- The output is true (equal to 1) when the input signal is greater than zero, and the previous value was less than or equal to zero.
- The output is false (equal to 0) when the input is negative or zero, or if the input is positive, the previous value was also positive.

Data Types: uint8 | Boolean

**Parameters**

**Initial condition — Initial condition of Boolean expression U/z > 0**
0 (default) | scalar | vector | matrix

Set the initial condition of the Boolean expression \( U/z > 0 \).

**Programmatic Use**

**Block Parameter:** vinit

**Type:** character vector

**Values:** scalar | vector | matrix

**Default:** '0'

**Input processing — Specify sample- or frame-based processing**

Elements as channels (sample based) (default) | Columns as channels (frame based)

Specify whether the block performs sample- or frame-based processing:

- **Columns as channels (frame based)** — Treat each column of the input as a separate channel (frame-based processing).

**Note** Frame-based processing requires a DSP System Toolbox license.
For more information, see “Sample- and Frame-Based Concepts” (DSP System Toolbox).

- **Elements as channels (sample based)** — Treat each element of the input as a separate channel (sample-based processing).

Use **Input processing** to specify whether the block performs sample- or frame-based processing. The block accepts frame-based signals for the input \( u \). All other input signals must be sample-based.

<table>
<thead>
<tr>
<th>Input Signal ( u )</th>
<th>Input Processing Mode</th>
<th>Block Works?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sample based</td>
<td>Sample based</td>
<td>Yes</td>
</tr>
<tr>
<td>Frame based</td>
<td></td>
<td>No, produces an error</td>
</tr>
<tr>
<td>Sample based</td>
<td>Frame based</td>
<td>Yes</td>
</tr>
<tr>
<td>Frame based</td>
<td></td>
<td>Yes</td>
</tr>
</tbody>
</table>

For more information about these two processing modes, see “Sample- and Frame-Based Concepts” (DSP System Toolbox).

**Programmatic Use**

**Block Parameter:** InputProcessing  
**Type:** character vector  
**Values:** 'Columns as channels (frame based)' | 'Elements as channels (sample based)'  
**Default:** 'Elements as channels (sample based)'

**Output data type** — **Output data type**

boolean (default) | uint8

Specify the output data type as boolean or uint8.

**Programmatic Use**

**Block Parameter:** OutDataTypeStr  
**Type:** character vector  
**Values:** 'boolean' | 'uint8'  
**Default:** 'boolean'
Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>double</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

Generated code relies on `memcpy` or `memset` functions (`string.h`) under certain conditions.

PLC Code Generation
Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

See Also
Detect Change | Detect Decrease | Detect Fall Negative | Detect Fall Nonpositive | Detect Increase | Detect Rise Nonnegative
Introduced before R2006a
Difference

Calculate change in signal over one time step
Library: Simulink / Discrete

Description

The Difference block outputs the current input value minus the previous input value.

Ports

Input

Port_1 — Input signal
scalar | vector | matrix | N-D array

Input signal, specified as a scalar, vector, matrix, or N-D array.

Dependencies

When you set Input processing to Columns as channels (frame based), the input signal must have two dimensions or less.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point

Output

Port_1 — Current input minus previous input
scalar | vector | matrix | N-D array

Current input minus previous input, specified as a scalar, vector, matrix, or N-D array.
Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point

Parameters

Main

Initial condition for previous input — Initial condition
0.0 (default) | scalar | vector | matrix | N-D array

Set the initial condition for the previous input.

Programmatic Use
Parameter: ICPrevInput
Type: character vector
Values: scalar | vector | matrix | N-D array
Default: '0.0'

Input processing — Specify sample- or frame-based processing
Elements as channels (sample based) (default) | Columns as channels (frame based)

Specify whether the block performs sample- or frame-based processing:

• Columns as channels (frame based) — Treat each column of the input as a separate channel (frame-based processing).

Note Frame-based processing requires a DSP System Toolbox license.

For more information, see “Sample- and Frame-Based Concepts” (DSP System Toolbox).

• Elements as channels (sample based) — Treat each element of the input as a separate channel (sample-based processing).

Use Input processing to specify whether the block performs sample- or frame-based processing. The block accepts frame-based signals for the input u. All other input signals must be sample-based.
Input Signal $u$ | Input Processing Mode | Block Works?
---|---|---
Sample based | Sample based | Yes
Frame based | | No, produces an error
Sample based | Frame based | Yes
Frame based | | Yes

For more information about these two processing modes, see “Sample- and Frame-Based Concepts” (DSP System Toolbox).

**Programmatic Use**

**Block Parameter:** InputProcessing

**Type:** character vector

**Values:** 'Columns as channels (frame based)' | 'Elements as channels (sample based)'

**Default:** 'Elements as channels (sample based)'

**Signal Attributes**

**Output minimum — Minimum output value for range checking**

```
[] (default) | scalar
```

Lower value of the output range that Simulink checks.

Simulink uses the minimum to perform:

- Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters”) for some blocks.
- Simulation range checking (see “Signal Ranges” and “Enable Simulation Range Checking”).
- Automatic scaling of fixed-point data types.
- Optimization of the code that you generate from the model. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. For more information, see “Optimize using the specified minimum and maximum values” (Simulink Coder).

**Note** Output minimum does not saturate or clip the actual output signal. Use the Saturation block instead.
Programmatic Use

Block Parameter: OutMin
Type: character vector
Values: ['[ ]'| scalar
Default: '[ ]'

Output maximum — Maximum output value for range checking
[] (default) | scalar

Upper value of the output range that Simulink checks.

Simulink uses the maximum value to perform:

- Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters”) for some blocks.
- Simulation range checking (see “Signal Ranges” and “Enable Simulation Range Checking”).
- Automatic scaling of fixed-point data types.
- Optimization of the code that you generate from the model. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. For more information, see “Optimize using the specified minimum and maximum values” (Simulink Coder).

Note: Output maximum does not saturate or clip the actual output signal. Use the Saturation block instead.

Programmatic Use

Block Parameter: OutMax
Type: character vector
Values: ['[ ]'| scalar
Default: '[ ]'

Output data type — Output data type
Inherit: Inherit via internal rule (default) | Inherit via back propagation | double | single | int8 | uint8 | int16 | uint16 | int32 | uint32 | int64 | uint64 | fixdt(1,16) | fixdt(1,16,0) | fixdt(1,16,2^0,0) | <data type expression>

Specify the output data type. You can set it to:
• A rule that inherits a data type, for example, Inherit: Inherit via back propagation
• The name of a built-in data type, for example, single
• The name of a data type object, for example, a Simulink.NumericType object
• An expression that evaluates to a data type, for example, fixdt(1,16,0)

Click the **Show data type assistant** button to display the **Data Type Assistant**, which helps you set the data type attributes. For more information, see “Specify Data Types Using Data Type Assistant”.

**Programmatic Use**
**Parameter:** OutDataTypeStr
**Type:** character vector
**Values:** 'Inherit: Inherit via internal rule' | 'Inherit: Inherit via back propagation' | 'double' | 'single' | 'int8' | 'uint8' | 'int16' | 'uint16' | 'int32' | 'uint32' | 'int64' | 'uint64' | 'boolean' | 'fixdt(1,16)' | 'fixdt(1,16,0)' | 'fixdt(1,16,2^0,0)' | '<data type expression>'
**Default:** 'Inherit: Inherit via internal rule'

**Lock output data type setting against changes by the fixed-point tools — Prevent fixed-point tools from overriding Output data type**
off (default) | on

Select this parameter to prevent the fixed-point tools from overriding the **Output** data type you specify on the block. For more information, see “Use Lock Output Data Type Setting” (Fixed-Point Designer).

**Programmatic Use**
**Block Parameter:** LockScale
**Type:** character vector
**Values:** 'off' | 'on'
**Default:** 'off'

**Integer rounding mode — Rounding mode for fixed-point operations**
Floor (default) | Ceiling | Convergent | Nearest | Round | Simplest | Zero

Specify the rounding mode for fixed-point operations. For more information, see “Rounding” (Fixed-Point Designer).
Block parameters always round to the nearest representable value. To control the rounding of a block parameter, enter an expression using a MATLAB rounding function into the mask field.

**Programmatic Use**

**Block Parameter:** RndMeth  
**Type:** character vector  
**Values:** 'Ceiling' | 'Convergent' | 'Floor' | 'Nearest' | 'Round' | 'Simplest' | 'Zero'  
**Default:** 'Floor'

Saturate to max or min when overflows occur — Method of overflow action  
off (default) | on

When you select this check box, overflows saturate to the maximum or minimum value that the data type can represent. Otherwise, overflows wrap.

When you select this check box, saturation applies to every internal operation on the block, not just the output or result. In general, the code generation process can detect when overflow is not possible. In this case, the code generator does not produce saturation code.

**Programmatic Use**

**Block Parameter:** DoSatur  
**Type:** character vector  
**Values:** 'off' | 'on'  
**Default:** 'off'

### Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>double</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Zero-Crossing Detection | no
---|---

a. This block is not recommended for use with Boolean signals.

### Extended Capabilities

#### C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

- The code generator does not explicitly group primitive blocks that constitute a nonatomic masked subsystem block in the generated code. This flexibility allows for more efficient code generation. In certain cases, you can achieve grouping by configuring the masked subsystem block to execute as an atomic unit by selecting the Treat as atomic unit option.
- Not recommended for production code. Relates to resource limits and restrictions on speed and memory often found in embedded systems. Generated code can contain dynamic allocation and freeing of memory, recursion, additional memory overhead, and widely-varying execution times. While the code is functionally valid and generally acceptable in resource-rich environments, smaller embedded targets often cannot support such code. Usually, blocks evolve toward being suitable for production code. Thus, blocks suitable for production code remain suitable.

#### PLC Code Generation
Generate Structured Text code using Simulink® PLC Coder™.

#### Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

### See Also
diff
Topics
“Sample- and Frame-Based Concepts” (DSP System Toolbox)

Introduced before R2006a
Digital Clock

Output simulation time at specified sampling interval

Library: Simulink / Sources

Description

The Digital Clock block outputs the simulation time only at the specified sampling interval. At other times, the block holds the output at the previous value. To control the precision of this block, use the Sample time parameter in the block dialog box.

Use this block rather than the Clock block (which outputs continuous time) when you need the current simulation time within a discrete system.

Ports

Output

Port_1 — Sample time
scalar

Sample time, in seconds, at the specified sampling interval. At other times, the block holds the output at the previous value.

Data Types: double

Parameters

Sample time — Sampling interval
1 (default) | scalar | vector
Specify the sampling interval in seconds. You can specify the sampling interval in one of two ways:

- As the period, specified as a real-valued scalar with data type double.
- As the period and offset, specified as a real-valued vector of length 2 with data type double. The period and offset must be finite and non-negative, and the offset value must be less than the period.

For more information, see Specifying Sample Time.

**Tip** Do not specify a continuous sample time, either 0 or [0,0]. Also, avoid specifying -1 (inheriting the sample time) because this block is a source.

**Programmatic Use**

Block Parameter: SampleTime  
Type: character vector  
Values: scalar | vector  
Default: '1'

**Block Characteristics**

<table>
<thead>
<tr>
<th>Data Types</th>
<th>double</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>no</td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
</tr>
</tbody>
</table>
Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

Not recommended for production code. Relates to resource limits and restrictions on speed and memory often found in embedded systems. Generated code can contain dynamic allocation and freeing of memory, recursion, additional memory overhead, and widely-varying execution times. While the code is functionally valid and generally acceptable in resource-rich environments, smaller embedded targets often cannot support such code. Usually, blocks evolve toward being suitable for production code. Thus, blocks suitable for production code remain suitable.

See Also
Clock

Topics
“Sample Time”

Introduced before R2006a
Direct Lookup Table (n-D)

Index into n-dimensional table to retrieve element, vector, or 2-D matrix

Library: Simulink / Lookup Tables
       HDL Coder / Lookup Tables

Description

The Direct Lookup Table (n-D) block indexes into an n-dimensional table to retrieve an element, vector, or 2-D matrix. The first selection index corresponds to the top (or left) input port. You can choose to provide the table data as an input to the block, or define the table data on the block dialog box. The number of input ports and the size of the output depend on the number of table dimensions and the output slice you select.

If you select a vector from a 2-D table, the output vector can be a column or a row, depending on the model configuration parameter setting Math and Data Types > Use algorithms optimized for row-major array layout. The block inputs are zero-based indices (for more information, see the Inputs select this object from table parameter.

The Direct Lookup Table block supports symbolic dimensions.

Block Inputs and Outputs

The Direct Lookup Table (n-D) block uses inputs as zero-based indices into an n-dimensional table. The number of inputs varies with the shape of the output: an element, vector, or 2-D matrix.

You define a set of output values as the Table data parameter. For the default column-major algorithm behavior, the first input specifies the zero-based index to the table dimension that is one higher than the output dimensionality. The next input specifies the zero-based index to the next table dimension, and so on.
### Output Shape

<table>
<thead>
<tr>
<th>Output Shape</th>
<th>Output Dimensionality</th>
<th>Table Dimension that Maps to the First Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>Element</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Vector</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Matrix</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

Suppose that you want to select a vector of values from a 4-D table.

The following mapping of block input port to table dimension applies.

<table>
<thead>
<tr>
<th>This input port...</th>
<th>Is the index for this table dimension...</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

## Changes in Block Icon Appearance

Depending on parameters you set, the block icon changes appearance. For table dimensions higher than 4, the icon matches the 4-D version but shows the exact number of dimensions at the top.

When you use the **Table data** parameter, you see these icons for the default column-major behavior. Some icons are different when you select the configuration parameter **Math and Data Types > Use algorithms optimized for row-major array layout.**
<table>
<thead>
<tr>
<th>Object that Inputs Select from the Table</th>
<th>Number of Table Dimensions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Element</td>
<td><img src="image1" alt="1-D T[k]" /></td>
</tr>
<tr>
<td>Vector</td>
<td><img src="image5" alt="1-D T[k]" /></td>
</tr>
<tr>
<td>2-D Matrix</td>
<td>Not applicable</td>
</tr>
</tbody>
</table>

When you use the table input port, you see these icons.
### Ports

**Input**

**Port 1 — Index i1 input values**

*scalar | vector*

For the default column-major algorithm, the first input port, specifying the zero-based index to the table dimension that is one higher than the output dimensionality (0, 1, or 2). The next input specifies the zero-based index to the next table dimension, and so on. All index inputs must be real-valued.

Data Types: *single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated*

**Port N — Index N input values**

*scalar | vector*

For the default column-major algorithm, the N-th input port, specifying the zero-based index to the table dimension that is N higher than the output dimensionality (0, 1, or 2). The number of inputs varies with the shape of the output. All index inputs must be real-valued.

Data Types: *single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated*

**T — Table data**

*vector | matrix | N-D array*
Table data, specified as a vector, matrix, or N-D array. The table size must match the dimensions of the **Number of dimensions** parameter. The block's output data type is the same as the table data type.

**Dependencies**

To enable this port, select the **Make table an input** check box.

**Data Types:** single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated

**Output**

**Port_1 — Output element, vector, or 2-D matrix**

scalar | vector | 2-D matrix

Output slice, provided as a scalar, vector, or 2-D matrix. The size of the block output is determined by the setting of the **Inputs select this object from table** parameter. The output data type is the same as the table data type.

**Data Types:** single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated

**Parameters**

**Main**

**Table**

**Number of table dimensions — Number of dimensions of table data**

2 (default) | 1 | 3 | 4

Number of dimensions that the **Table data** parameter must have. This value determines the number of independent variables for the table and the number of inputs to the block.

<table>
<thead>
<tr>
<th>To specify...</th>
<th>Do this...</th>
</tr>
</thead>
<tbody>
<tr>
<td>1, 2, 3, or 4</td>
<td>Select the value from the drop-down list.</td>
</tr>
</tbody>
</table>
To specify... | Do this...
---|---
A higher number of table dimensions | Enter a positive integer directly in the field.
The maximum number of table dimensions that this block supports is 30.

**Programmatic Use**

**Block Parameter:** NumberOfTableDimensions  
**Type:** character vector  
**Values:** '1' | '2' | '3' | '4' | ... | '30' |  
**Default:** '2'

**Make table an input — Provide table data as a block input**

off (default) | on

Select this check box to provide table data to the Direct Lookup Table (n-D) block as a block input. When you select this check box, a new input port, T, appears. Use this port to input the table data.

**Programmatic Use**

**Block Parameter:** TableIsInput  
**Type:** character vector  
**Values:** 'off' | 'on'  
**Default:** 'off'

**Table data — Table of output values**

[4 5 6; 16 19 20; 10 18 23](default) | scalar, vector, matrix, or N-D array

Specify the table of output values. The table size must match the dimensions of the **Number of table dimensions** parameter.

| **Tip** | During block diagram editing, you can leave the **Table data** field empty. But for simulation, you must match the number of dimensions in **Table data** to the **Number of table dimensions**. For details on how to construct multidimensional MATLAB arrays, see “Multidimensional Arrays” (MATLAB). |

Click **Edit** to open the Lookup Table Editor. For more information, see “Edit Lookup Tables”.

1-443
Dependencies

To enable the Table data field, clear the Make table an input check box.

Programmatic Use

Block Parameter: Table
Type: character vector
Values: scalar, vector, matrix, or N-D array
Default: ' [4 5 6; 16 19 20; 10 18 23]' 

Algorithm

Inputs select this object from table — Specify whether output is an element, vector, or 2-D matrix

Element (default) | Vector | 2-D Matrix

Specify whether the output data is a single element, a vector, or a 2-D matrix. The number of input ports for indexing depends on your selection.

<table>
<thead>
<tr>
<th>Selection</th>
<th>Number of Input Ports for Indexing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Element</td>
<td>Number of table dimensions</td>
</tr>
<tr>
<td>Vector</td>
<td>Number of table dimensions -1</td>
</tr>
<tr>
<td>2-D Matrix</td>
<td>Number of table dimensions -2</td>
</tr>
</tbody>
</table>

This numbering matches MATLAB indexing. For example, if you have a 4-D table of data, follow these guidelines.

<table>
<thead>
<tr>
<th>To access...</th>
<th>Specify...</th>
<th>As in...</th>
</tr>
</thead>
<tbody>
<tr>
<td>An element</td>
<td>Four indices</td>
<td>array(1,2,3,4)</td>
</tr>
<tr>
<td>A vector</td>
<td>Three indices</td>
<td>array(:,2,3,4) (default column-major algorithm)</td>
</tr>
<tr>
<td>A 2-D matrix</td>
<td>Two indices</td>
<td>array(:, :, 3,4) (default column-major algorithm)</td>
</tr>
</tbody>
</table>

Tips

When the Math and Data Types > Use algorithms optimized for row-major array layout configuration parameter is set, the Direct Lookup Table block behavior changes from column-major to row-major. For this block, the column-major and row-major algorithms may differ semantically in output calculations, resulting in different numerical
values. For example, assume that **Inputs select this object from table** parameter is set to Vector. The elements of the selected vector are contiguous in the table storage memory. This table shows the column-major and row-major algorithm depending on the table dimension:

<table>
<thead>
<tr>
<th>Table Dimension</th>
<th>Column-Major Algorithm</th>
<th>Row-Major Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-D table</td>
<td>Column vector is selected</td>
<td>Row vector is selected</td>
</tr>
<tr>
<td>3-D and higher table</td>
<td>Output vector is selected from the first dimension of the table</td>
<td>Output vector is selected from the last dimension of the table</td>
</tr>
</tbody>
</table>

Consider the row-major and column-major direct lookup algorithms with vector output from a 3-D table. The last dimension is the third dimension of a 3-D table. Due to semantic changes, column-major and row-major direct lookup may output different vector size and numerical values.

This figure shows a Direct Lookup Table (n-D) block configured with a 3-D table and a vector output. By default, the block icon shows the column-major algorithm.

To have the same block use the row-major algorithm, change the **Math and Data Types > Use algorithm optimized for row-major layout** configuration parameter of the model and recompile. The block icon changes to reflect the change to the algorithm optimized for row-major behavior.
For more information on row-major support, see “Row-Major Array Layout: Simplify integration with external C/C++ code for Lookup Table and other blocks” (Simulink Coder).

**Programmatic Use**

**Block Parameter:** InputsSelectThisObjectFromTable  
**Type:** character vector  
**Values:** 'Element' | 'Vector' | '2-D Matrix'  
**Default:** 'Element'

**Diagnostic for out-of-range input — Block action when input is out of range**  
**Warning (default) | None | Error**

Specify whether to show a warning or error when an index is out of range with respect to the table dimension. Options include:

- **None** — Produce no response.
- **Warning** — Display a warning and continue the simulation.
- **Error** — Terminate the simulation and display an error.

When you select None or Warning, the block clamps out-of-range indices to fit table dimensions. For example, if the specified index is 5.3 and the maximum index for that table dimension is 4, the block clamps the index to 4.

**Programmatic Use**

**Block Parameter:** DiagnosticForOutOfRangeInput  
**Type:** character vector  
**Values:** 'None' | 'Warning' | 'Error'  
**Default:** 'Warning'

**Sample time — Specify sample time as a value other than -1**  
-1 (default) | scalar | vector

Specify the sample time as a value other than -1. For more information, see “Specify Sample Time”.

**Dependencies**

This parameter is not visible unless it is explicitly set to a value other than -1. To learn more, see “Blocks for Which Sample Time Is Not Recommended”.

1-446
Programmatic Use
Block Parameter: SampleTime
Type: character vector
Values: scalar or vector
Default: '-1'

Table Attributes

Note The parameters in the Table Attributes pane are not available if you select Make table an input. In this case, the block inherits all table attributes from the input port with the label T.

Table minimum — Minimum value table data can have
[] (default) | finite, real, double, scalar

Specify the minimum value for table data. The default value is [] (unspecified).

Programmatic Use
Block Parameter: TableMin
Type: character vector
Values: scalar
Default: '[]'

Table maximum — Maximum value table data can have
[] (default) | finite, real, double, scalar

Specify the maximum value for table data. The default value is [] (unspecified).

Programmatic Use
Block Parameter: TableMax
Type: character vector
Values: scalar
Default: '[]'

Table data type — Data type of table data
Inherit: Inherit from 'Table data' (default) | double | single | int8 | uint8 | int16 | uint16 | int32 | uint32 | boolean | fixdt(1,16) | fixdt(1,16,0) | fixdt(1,16,2^0,0) | Enum: <class name> | <data type expression>

Specify the table data type. You can set it to:
• A rule that inherits a data type, for example, Inherit: Inherit from 'Table data'
• The name of a built-in data type, for example, single
• The name of a data type class, for example, an enumerated data type class
• An expression that evaluates to a data type, for example, fixdt(1,16,0)

Click the Show data type assistant button to display the Data Type Assistant, which helps you set the data type attributes. For more information, see “Specify Data Types Using Data Type Assistant”.

**Programmatic Use**
**Block Parameter:** TableDataTypeStr
**Type:** character vector
**Values:** 'Inherit: Inherit from 'Table data'' | 'double' | 'single' | 'int8' | 'uint8' | 'int16' | 'uint16' | 'int32' | 'uint32' | boolean | 'fixdt(1,16)' | 'fixdt(1,16,0)' | 'fixdt(1,16,2^0,0)'|'Enum: <class name>'|'<data type expression>'
**Default:** 'Inherit: Inherit from 'Table data''

**Lock data type settings against changes by the fixed-point tools — Prevent fixed-point tools from overriding data types**
**off** (default) | **on**

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on this block. For more information, see “Lock the Output Data Type Setting” (Fixed-Point Designer).

**Programmatic Use**
**Block Parameter:** LockScale
**Type:** character vector
**Values:** 'off' | 'on'
**Default:** 'off'

**Block Characteristics**

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>double</th>
<th>enumerated</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
</table>


Direct Feedthrough | yes
Multidimensional Signals | yes
Variable-Size Signals | no
Zero-Crossing Detection | no

a. This block supports fixed-point data types for 'Table' data only.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture
This block has a single, default HDL architecture.

HDL Block Properties

| ConstrainedOutput Pipeline | Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. For more details, see “ConstrainedOutputPipeline” (HDL Coder). |
### InputPipeline
Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “InputPipeline” (HDL Coder).

### OutputPipeline
Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “OutputPipeline” (HDL Coder).

#### Restrictions

**MAX 10 Device Settings**

If you use Intel® MAX 10 device, to map the lookup table to RAM, add this Tcl command when creating the project in the Quartus tool:

```tcl
set_global_assignment -name INTERNAL_FLASH_UPDATE_MODE "SINGLE IMAGE WITH ERAM"
```

**Required Block Settings**

- **Number of table dimensions**: HDL Coder supports a maximum dimension of 2.
- **Inputs select this object from table**: Select Element.
- **Make table an input**: Clear this check box.
- **Diagnostic for out-of-range input**: Select Error. If you select other options, the coder displays a warning.

**Table Data Typing and Sizing**

- It is good practice to size each dimension in the table to be a power of two. If the length of a dimension (except the innermost dimension) is not a power of two, HDL Coder issues a warning. By following this practice, you can avoid multiplications during table indexing operations and realize a more efficient table in hardware.
- Table data must resolve to a nonfloating-point data type. The coder examines the output port to verify that its data type meets this requirement.
- All ports on the block require scalar values.
Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

This block supports fixed-point data types for Table data only. In addition to built-in data types, the index input also supports fixed-point data type with bias equal to 0, slope equal to 1.0, fractional length equal to 0, and word length less than or equal to 128.

See Also
n-D Lookup Table

Topics
“About Lookup Table Blocks”
“Anatomy of a Lookup Table”
“Enter Breakpoints and Table Data”
“Guidelines for Choosing a Lookup Table”
“Direct Lookup Table Algorithm for Row-Major Array Layout” (Simulink Coder)
“Column-Major Layout to Row-Major Layout Conversion of Models with Lookup Table Blocks” (Simulink Coder)

Introduced before R2006a
Discrete Derivative

Compute discrete-time derivative
Library: Simulink / Discrete

Description

The Discrete Derivative block computes an optionally scaled discrete time derivative as follows

\[ y(t_n) = K \left( \frac{u(t_n) - u(t_n - 1)}{T_s} \right) \]

where

- \( u(t_n) \) and \( y(t_n) \) are the block input and output at the current time step, respectively.
- \( u(t_n - 1) \) is the block input at the previous time step.
- \( K \) is an optional scaling factor, specified using the Gain value parameter.
- \( T_s \) is the simulation's discrete step size, which must be fixed.

Note  Do not use this block in subsystems with a nonperiodic trigger (for example, nonperiodic function-call subsystems). This configuration produces inaccurate results.

Ports

Input

Port_1 — Input signal
scalar | vector | matrix
Input signal, specified as a scalar, vector, or matrix.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

**Output**

Port_1 — Scaled discrete time derivative
scalar | vector | matrix

Optionally scaled discrete-time derivative, specified as a scalar, vector, or matrix. For more information on how the block computes the discrete-time derivative, see “Description” on page 1-452. You specify the data type of the output signal with the **Output data type** parameter.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

**Parameters**

**Main**

Gain value — Scaling factor
1.0 (default) | scalar

Scaling factor applied to the computed derivative, specified as a real scalar value.

**Programmatic Use**

Block Parameter: gainval
Type: character vector
Values: scalar
Default: '1.0'

Initial condition for previous weighted input K*u/Ts — Initial condition
0.0 (default) | scalar

Initial condition for the previous scaled input, specified as a scalar.

**Programmatic Use**

Block Parameter: ICPrevScaledInput
Input processing — Specify sample- or frame-based processing

Elements as channels (sample based) (default) | Columns as channels (frame based)

Specify whether the block performs sample- or frame-based processing:

- **Columns as channels (frame based)** — Treat each column of the input as a separate channel (frame-based processing).

**Note** Frame-based processing requires a DSP System Toolbox license.

For more information, see “Sample- and Frame-Based Concepts” (DSP System Toolbox).

- **Elements as channels (sample based)** — Treat each element of the input as a separate channel (sample-based processing).

Use **Input processing** to specify whether the block performs sample- or frame-based processing. The block accepts frame-based signals for the input \( u \). All other input signals must be sample-based.

<table>
<thead>
<tr>
<th>Input Signal ( u )</th>
<th>Input Processing Mode</th>
<th>Block Works?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sample based</td>
<td>Sample based</td>
<td>Yes</td>
</tr>
<tr>
<td>Frame based</td>
<td></td>
<td>No, produces an error</td>
</tr>
<tr>
<td>Sample based</td>
<td>Frame based</td>
<td>Yes</td>
</tr>
<tr>
<td>Frame based</td>
<td></td>
<td>Yes</td>
</tr>
</tbody>
</table>

For more information about these two processing modes, see “Sample- and Frame-Based Concepts” (DSP System Toolbox).

**Programmatic Use**

**Block Parameter:** `InputProcessing`

**Type:** character vector

**Values:** 'Columns as channels (frame based)' | 'Elements as channels (sample based)'

**Default:** 'Elements as channels (sample based)'
**Signal Attributes**

**Output minimum — Minimum output value for range checking**

[] (default) | scalar

Lower value of the output range that Simulink checks.

Simulink uses the minimum to perform:

- Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters”) for some blocks.
- Simulation range checking (see “Signal Ranges” and “Enable Simulation Range Checking”).
- Automatic scaling of fixed-point data types.
- Optimization of the code that you generate from the model. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. For more information, see “Optimize using the specified minimum and maximum values” (Simulink Coder).

**Note** *Output minimum* does not saturate or clip the actual output signal. Use the Saturation block instead.

**Programmatic Use**

**Block Parameter:** OutMin  
**Type:** character vector  
**Values:** ' [ ] ' | scalar  
**Default:** ' [ ] '

**Output maximum — Maximum output value for range checking**

[] (default) | scalar

Upper value of the output range that Simulink checks.

Simulink uses the maximum value to perform:

- Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters”) for some blocks.
- Simulation range checking (see “Signal Ranges” and “Enable Simulation Range Checking”).
• Automatic scaling of fixed-point data types.
• Optimization of the code that you generate from the model. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. For more information, see “Optimize using the specified minimum and maximum values” (Simulink Coder).

**Note** Output maximum does not saturate or clip the actual output signal. Use the Saturation block instead.

**Programmatic Use**
**Block Parameter:** OutMax  
**Type:** character vector  
**Values:** '[]' | scalar  
**Default:** '[]'

**Output data type — Output data type**
Inherit: Inherit via internal rule (default) | Inherit: Inherit via back propagation | double | single | int8 | uint8 | int16 | uint16 | int32 | uint32 | int64 | uint64 | fixdt(1,16,0) | fixdt(1,16,2^0,0)

Specify the output data type. You can set it to:

• A rule that inherits a data type, for example, Inherit: Inherit via back propagation
• The name of a built-in data type, for example, single
• The name of a data type object, for example, a Simulink.NumericType object
• An expression that evaluates to a data type, for example, fixdt(1,16,0)

Click the **Show data type assistant** button to display the **Data Type Assistant**, which helps you set the data type attributes. For more information, see “Specify Data Types Using Data Type Assistant”.

**Programmatic Use**
**Block Parameter:** OutDataTypeStr  
**Type:** character vector  
**Values:** 'Inherit: Inherit via internal rule' | 'Inherit: Inherit via back propagation' | 'double' | 'single' | 'int8' | 'uint8' | 'int16' | 'uint16' | 'int32' | 'uint32' | 'int64' | 'uint64' | 'fixdt(1,16,0)' | 'fixdt(1,16,2^0,0)'
Lock output data type setting against changes by the fixed-point tools — Prevent fixed-point tools from overriding Output data type

Select this parameter to prevent the fixed-point tools from overriding the Output data type you specify on the block. For more information, see “Use Lock Output Data Type Setting” (Fixed-Point Designer).

Programmatic Use
Block Parameter: LockScale
Type: character vector
Values: 'off' | 'on'
Default: 'off'

Integer rounding mode — Rounding mode for fixed-point operations

Specify the rounding mode for fixed-point operations. For more information, see “Rounding” (Fixed-Point Designer).

Block parameters always round to the nearest representable value. To control the rounding of a block parameter, enter an expression using a MATLAB rounding function into the mask field.

Programmatic Use
Block Parameter: RndMeth
Type: character vector
Values: 'Ceiling' | 'Convergent' | 'Floor' | 'Nearest' | 'Round' | 'Simplest' | 'Zero'
Default: 'Floor'

Saturate to max or min when overflows occur — Method of overflow action

When you select this check box, overflows saturate to the maximum or minimum value that the data type can represent. Otherwise, overflows wrap.

When you select this check box, saturation applies to every internal operation on the block, not just the output or result. In general, the code generation process can detect
when overflow is not possible. In this case, the code generator does not produce saturation code.

**Programmatic Use**

**Block Parameter:** DoSatur  
**Type:** character vector  
**Values:** 'off' | 'on'  
**Default:** 'off'

### Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>double</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Extended Capabilities

#### C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

- Depends on absolute time when used inside a triggered subsystem hierarchy.
- Generated code relies on memcpy or memset functions (string.h) under certain conditions.

#### PLC Code Generation
Generate Structured Text code using Simulink® PLC Coder™.
Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

See Also
Derivative | Discrete-Time Integrator

Introduced before R2006a
Discrete Filter

Model Infinite Impulse Response (IIR) filters

Library: Simulink / Discrete

Description

The Discrete Filter block independently filters each channel of the input signal with the specified digital IIR filter. You can specify the filter structure as Direct form I, Direct form I transposed, Direct form II, or Direct form II transposed. The block implements static filters with fixed coefficients. You can tune the coefficients of these static filters.

This block filters each channel of the input signal independently over time. The Input processing parameter allows you to specify how the block treats each element of the input. You can specify treating input elements as an independent channel (sample-based processing), or treating each column of the input as an independent channel (frame-based processing). To perform frame-based processing, you must have a DSP System Toolbox license.

The output dimensions equal the input dimensions, except when you specify a matrix of filter taps for the Numerator coefficients parameter. When you do so, the output dimensions depend on the number of different sets of filter taps you specify.

Use the Numerator coefficients parameter to specify the coefficients of the discrete filter numerator polynomial. Use the Denominator coefficients parameter to specify the coefficients of the denominator polynomial of the function. The Denominator coefficients parameter must be a vector of coefficients.

Specify the coefficients of the numerator and denominator polynomials in ascending powers of $z^{-1}$. The Discrete Filter block lets you use polynomials in $z^{-1}$ (the delay operator) to represent a discrete system. Signal processing engineers typically use this method. Conversely, the Discrete Transfer Fcn block lets you use polynomials in $z$ to represent a
discrete system. Control engineers typically use this method. When the numerator and denominator polynomials have the same length, the two methods are identical.

**Specifying Initial States**

In Dialog parameters and Input port(s) modes, the block initializes the internal filter states to zero by default, which is equivalent to assuming past inputs and outputs are zero. You can optionally use the **Initial states** parameter to specify nonzero initial states for the filter delays.

To determine the number of initial state values you must specify, and how to specify them, see the following table on valid initial states and Number of Delay Elements (Filter States). The **Initial states** parameter can take one of four forms as described in the following table.
### Valid Initial States

<table>
<thead>
<tr>
<th>Initial state</th>
<th>Examples</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scalar</td>
<td>5</td>
<td>Each delay element for each channel is set to 5. The block initializes all delay elements in the filter to the scalar value.</td>
</tr>
<tr>
<td>Vector (for applying the same delay elements to each channel)</td>
<td>For a filter with two delay elements: [d₁d₂] [d₁ D₁ d₁] or [d₂ D₂ d₂]</td>
<td>Each vector element specifies a unique initial condition for a corresponding delay element. The block applies the same vector of initial conditions to each channel of the input signal. The vector length must equal the number of delay elements in the filter (specified in the table Number of Delay Elements (Filter States)).</td>
</tr>
<tr>
<td>Vector or matrix (for applying different delay elements to each channel)</td>
<td>For a three-channel input signal and a filter with two delay elements: [d₁d₂D₁D₂d₁d₂]</td>
<td>Each vector or matrix element specifies a unique initial condition for a corresponding delay element in a corresponding channel:</td>
</tr>
<tr>
<td></td>
<td>[d₁ D₁ d₁]</td>
<td>• The vector length must be equal to the product of the number of input channels and the number of delay elements in the filter (specified in the table Number of Delay Elements (Filter States)).</td>
</tr>
<tr>
<td></td>
<td>[d₂ D₂ d₂]</td>
<td>• The matrix must have the same number of rows as the number of delay elements in the filter (specified in the table Number of Delay Elements (Filter States)), and must have one column for each channel of the input signal.</td>
</tr>
<tr>
<td>Empty matrix</td>
<td>[ ]</td>
<td>The empty matrix, [ ], is equivalent to setting the Initial conditions parameter to the scalar value 0.</td>
</tr>
</tbody>
</table>

The number of delay elements (filter states) per input channel depends on the filter structure, as indicated in the following table.
Number of Delay Elements (Filter States)

<table>
<thead>
<tr>
<th>Filter Structure</th>
<th>Number of Delay Elements Per Channel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct form I</td>
<td>• number of zeros - 1</td>
</tr>
<tr>
<td>Direct form I transposed</td>
<td>• number of poles - 1</td>
</tr>
<tr>
<td>Direct form II</td>
<td>max(number of zeros, number of poles) - 1</td>
</tr>
<tr>
<td>Direct form II transposed</td>
<td></td>
</tr>
</tbody>
</table>

The following tables describe the valid initial states for different sizes of input and different number of channels depending on whether you set the **Input processing** parameter to frame based or sample based.

**Frame-Based Processing**

<table>
<thead>
<tr>
<th>Input</th>
<th>Number of Channels</th>
<th>Valid Initial States (Dialog Box)</th>
<th>Valid Initial States (Input Port)</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Column vector (K-by-1)</td>
<td>1</td>
<td>• Scalar</td>
<td>• Scalar</td>
</tr>
<tr>
<td>• Unoriented vector (K)</td>
<td></td>
<td>• Column vector (M-by-1)</td>
<td>• Column vector (M-by-1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Row vector (1-by-M)</td>
<td>• Row vector (1-by-M)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Row vector (1-by-N)</td>
<td>N</td>
<td>• Scalar</td>
<td>• Scalar</td>
</tr>
<tr>
<td>• Matrix (K-by-N)</td>
<td></td>
<td>• Column vector (M-by-1)</td>
<td>• Matrix (M-by-N)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Row vector (1-by-M)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Matrix (M-by-N)</td>
<td></td>
</tr>
</tbody>
</table>
Sample-Based Processing

<table>
<thead>
<tr>
<th>Input</th>
<th>Number of Channels</th>
<th>Valid Initial States (Dialog Box)</th>
<th>Valid Initial States (Input Port)</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Scalar</td>
<td>1</td>
<td>• Scalar</td>
<td>• Scalar</td>
</tr>
<tr>
<td>• Row vector (1-by-(N))</td>
<td>(N)</td>
<td>• Column vector ((M)-by-1)</td>
<td>• Column vector ((M)-by-1)</td>
</tr>
<tr>
<td>• Column vector ((N)-by-1)</td>
<td></td>
<td>• Row vector (1-by-(M))</td>
<td>• Row vector (1-by-(M))</td>
</tr>
<tr>
<td>• Unoriented vector ((N))</td>
<td></td>
<td>• Matrix ((M)-by-N)</td>
<td>• Scalar</td>
</tr>
<tr>
<td>• Matrix ((K)-by-(N))</td>
<td>(K \times N)</td>
<td>• Scalar</td>
<td>• Scalar</td>
</tr>
<tr>
<td>• Column vector ((M)-by-1)</td>
<td></td>
<td>• Column vector ((M)-by-1)</td>
<td></td>
</tr>
<tr>
<td>• Row vector (1-by-(M))</td>
<td></td>
<td>• Matrix ((M)-by-((K\times N)))</td>
<td></td>
</tr>
</tbody>
</table>

When the **Initial states** is a scalar, the block initializes all filter states to the same scalar value. Enter 0 to initialize all states to zero. When the **Initial states** is a vector or a matrix, each vector or matrix element specifies a unique initial state. This unique state corresponds to a delay element in a corresponding channel:

- The vector length must equal the number of delay elements in the filter, \(M = \max(\text{number of zeros}, \text{number of poles})\).
- The matrix must have the same number of rows as the number of delay elements in the filter, \(M = \max(\text{number of zeros}, \text{number of poles})\). The matrix must also have one column for each channel of the input signal.

The following example shows the relationship between the initial filter output and the initial input and state. Given an initial input \(u_1\), the first output \(y_1\) is related to the initial state \([x_1, x_2]\) and initial input by:
\[ y_1 = b_1 \frac{(u_1 - a_2 x_1 - a_3 x_2)}{a_1} + b_2 x_1 + b_3 x_2 \]

**Ports**

**Input**

\[ u \] — Input signal  
Scalar | Vector | Matrix

Input signal to filter, specified as a scalar, vector, or matrix.

**Dependencies**

The name of this port depends on the source you specify for the numerator coefficients, denominator coefficients and initial states. When you set **Numerator**, **Denominator**, and **Initial states** to Dialog, there is only one input port, and the port is unlabeled. When you set **Numerator**, **Denominator**, or **Initial states** to Input port, this port is labeled \[ u \].

Data Types: Single | Double | Int8 | Int16 | Int32 | Fixed Point

**Num** — Numerator coefficients  
Scalar | Vector | Matrix

Numerator coefficients of the discrete filter, specified as descending powers of \( z \). Use a row vector to specify the coefficients for a single numerator polynomial.
Dependencies

To enable this port, set Numerator to Input port.

Data Types: single | double | int8 | int16 | int32 | fixed point

Den — Denominator coefficients
scalar | vector

Specify the denominator coefficients of the discrete filter as descending powers of z. Use a row vector to specify the coefficients for a single denominator polynomial.

Dependencies

To enable this port, set Denominator to Input port.

Data Types: single | double | int8 | int16 | int32 | fixed point

x0 — Initial states
scalar | vector | matrix

Initial states, specified as a scalar, vector, or matrix. For more information about specifying states, see “Specifying Initial States” on page 1-461.

Dependencies

To enable this port, set the Filter structure to Direct form II or Direct form II transposed, and set Initial states to Input port.

Data Types: single | double | int8 | int16 | int32 | fixed point

Output

Port_1 — Filtered output signal
scalar | vector | matrix

Filtered output signal. The output dimensions equal the input dimensions, except when you specify a matrix of filter taps for the Numerator coefficients parameter. When you do so, the output dimensions depend on the number of different sets of filter taps you specify.

Data Types: single | double | int8 | int16 | int32 | fixed point
Parameters

Main

Filter structure — Filter structure
Direct form II (default) | Direct form I transposed | Direct form I | Direct form II transposed

Specify the discrete IIR filter structure.

Dependencies

To use any filter structure other than Direct form II, you must have an available DSP System Toolbox license.

Programmatic Use

Block Parameter: FilterStructure
Type: character vector
Values: 'Direct form II' | 'Direct form I transposed' | 'Direct form I' | 'Direct form II transposed'
Default: 'Direct form II'

Numerator Source — Source of numerator coefficients
Dialog (default) | Input port

Specify the source of the numerator coefficients as Dialog or Input port.

Programmatic Use

Block Parameter: NumeratorSource
Type: character vector
Values: 'Dialog' | 'Input port'
Default: 'Dialog'

Numerator Value — Numerator coefficients
[1] (default) | scalar | vector | matrix

Specify the numerator coefficients of the discrete filter as descending powers of z. Use a row vector to specify the coefficients for a single numerator polynomial.

Dependencies

To enable this parameter, set the Numerator Source to Dialog.
Programmatic Use
Block Parameter: Numerator
Type: character vector
Values: scalar | vector | matrix
Default: '[1]'

Denominator Source — Source of denominator coefficients
Dialog (default) | Input port

Specify the source of the denominator coefficients as Dialog or Input port.

Programmatic Use
Block Parameter: DenominatorSource
Type: character vector
Values: 'Dialog' | 'Input port'
Default: 'Dialog'

Denominator Value — Denominator coefficients
[1 0.5] (default) | vector

Specify the denominator coefficients of the discrete filter as descending powers of z. Use a row vector to specify the coefficients for a single denominator polynomial.

Dependencies
To enable this parameter, set the Denominator Source to Dialog.

Programmatic Use
Block Parameter: Denominator
Type: character vector
Values: scalar | vector
Default: '[1 0.5]'

Initial states Source — Source of initial states
Dialog (default) | Input port

Specify the source of the initial states as Dialog or Input port.

Programmatic Use
Block Parameter: InitialStatesSource
Type: character vector
Values: 'Dialog' | 'Input port'
Default: 'Dialog'
Initial states Value — Initial filter states
0 (default) | scalar | vector | matrix

Specify the initial filter states as a scalar, vector, or matrix. To learn how to specify initial states, see “Specifying Initial States” on page 1-461.

Dependencies

To enable this parameter, set the Filter structure to Direct form II or Direct form II transposed, and set Initial states Source to Dialog.

Programmatic Use
Block Parameter: InitialStates
Type: character vector
Values: scalar | vector | matrix
Default: '0'

Initial states on numerator side — Initial numerator states
0 (default) | scalar | vector | matrix

Specify the initial numerator filter states as a scalar, vector, or matrix. To learn how to specify initial states, see “Specifying Initial States” on page 1-461.

Dependencies

To enable this port, set the Filter structure to Direct form I or Direct form I transposed.

Programmatic Use
Block Parameter: InitialStates
Type: character vector
Values: scalar | vector | matrix
Default: '0'

Initial states on denominator side — Initial denominator states
0 (default) | scalar | vector | matrix

Specify the initial denominator filter states as a scalar, vector, or matrix. To learn how to specify initial states, see “Specifying Initial States” on page 1-461.

Dependencies

To enable this port, set the Filter structure to Direct form I or Direct form I transposed.
Programmatic Use
Block Parameter: InitialDenominatorStates
Type: character vector
Values: scalar | vector | matrix
Default: '0'

External reset — External state reset
None (default) | Rising | Falling | Either | Level | Level hold

Specify the trigger event to use to reset the states to the initial conditions.

<table>
<thead>
<tr>
<th>Reset Mode</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>No reset</td>
</tr>
<tr>
<td>Rising</td>
<td>Reset on a rising edge</td>
</tr>
<tr>
<td>Falling</td>
<td>Reset on a falling edge</td>
</tr>
<tr>
<td>Either</td>
<td>Reset on either a rising or falling edge</td>
</tr>
<tr>
<td>Level</td>
<td>Reset in either of these cases:</td>
</tr>
<tr>
<td></td>
<td>• When the reset signal is nonzero at the current time step</td>
</tr>
<tr>
<td></td>
<td>• When the reset signal value changes from nonzero at the previous time step to zero at the current time step</td>
</tr>
<tr>
<td>Level hold</td>
<td>Reset when the reset signal is nonzero at the current time step</td>
</tr>
</tbody>
</table>

Programmatic Use
Block Parameter: ExternalReset
Type: character vector
Values: 'None' | 'Rising' | 'Falling' | 'Either' | 'Level' | 'Level hold'
Default: 'None'

Input processing — Sample- or frame-based processing
Elements as channels (sample based) (default) | Columns as channels (frame based)

Specify whether the block performs sample- or frame-based processing.

• Elements as channels (sample based) — Process each element of the input as an independent channel.
• **Columns as channels (frame based)** — Process each column of the input as an independent channel.

**Dependencies**

Frame-based processing requires a DSP System Toolbox license.

For more information, see “Sample- and Frame-Based Concepts” (DSP System Toolbox).

**Programmatic Use**

**Block Parameter:** InputProcessing  
**Type:** character vector  
**Values:** 'Columns as channels (frame based)' | 'Elements as channels (sample based)'  
**Default:** 'Elements as channels (sample based)'

**Optimize by skipping divide by leading denominator coefficient \((a_0)\)**  
— **Skip divide by \(a_0\)**  
  
  off (default) | on

Select when the leading denominator coefficient, \(a_0\), equals one. This parameter optimizes your code.

When you select this check box, the block does not perform a divide-by-\(a_0\) either in simulation or in the generated code. An error occurs if \(a_0\) is not equal to one.

When you clear this check box, the block is fully tunable during simulation. It performs a divide-by-\(a_0\) in both simulation and code generation.

**Programmatic Use**

**Block Parameter:** a0EqualsOne  
**Type:** character vector  
**Values:** 'off' | 'on'  
**Default:** 'off'

**Sample time (-1 for inherited)** — **Interval between samples**

-1 (default) | scalar | vector

Specify the time interval between samples. To inherit the sample time, set this parameter to -1. For more information, see “Specify Sample Time”.

**Programmatic Use**

**Block Parameter:** SampleTime
**Type:** character vector  
**Values:** scalar | vector  
**Default:** '-1'

## Data Types

**State — State data type**  
*Inherit: Same as input* (default) | `int8` | `int16` | `int32` | `int64` | `fixdt(1,16,0)`  
| <data type expression>

Specify the state data type. You can set this parameter to:

- A rule that inherits a data type, for example, **Inherit: Same as input**
- A built-in integer, for example, `int8`
- A data type object, for example, a `Simulink.NumericType` object
- An expression that evaluates to a data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button to display the **Data Type Assistant**, which helps you set the data type attributes. For more information, see “Specify Data Types Using Data Type Assistant”.

### Programmatic Use

**Block Parameter:** `StateDataTypeStr`  
**Type:** character vector  
**Values:**  
- 'Inherit: Same as input'  
- 'int8'  
- 'int16'  
- 'int32'  
- 'int64'  
- 'fixdt(1,16,0)'  
- '<data type expression>'  
**Default:** 'Inherit: Same as input'

**Numerator coefficients — Numerator coefficient data type**  
*Inherit: Inherit via internal rule* (default) | `int8` | `int16` | `int32` | `int64` | `fixdt(1,16)` | `fixdt(1,16,0)`  
| <data type expression>

Specify the numerator coefficient data type. You can set this parameter to:

- A rule that inherits a data type, for example, **Inherit: Inherit via internal rule**
- A built-in signed integer, for example, `int8`
- A data type object, for example, a `Simulink.NumericType` object
• An expression that evaluates to a data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button to display the **Data Type Assistant**, which helps you set the data type attributes. For more information, see “Specify Data Types Using Data Type Assistant”.

**Programmatic Use**

**Block Parameter:** `NumCoeffDataTypeStr`  
**Type:** character vector  
**Values:** 'Inherit: Inherit via internal rule' | 'int8' | 'int16' | 'int32' | 'int64' | 'fixdt(1,16)' | 'fixdt(1,16,0)' | '<data type expression>'  
**Default:** 'Inherit: Inherit via internal rule'

**Numerator coefficient minimum — Minimum value of numerator coefficients**  
`[]` (default) | scalar

Specify the minimum value that a numerator coefficient can have. The default value is `[ ]` (unspecified). Simulink software uses this value to perform:

• Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters”)
• Automatic scaling of fixed-point data types

**Programmatic Use**

**Block Parameter:** `NumCoeffMin`  
**Type:** character vector  
**Values:** scalar  
**Default:** `['']`

**Numerator coefficient maximum — Maximum value of numerator coefficients**  
`[]` (default) | scalar

Specify the maximum value that a numerator coefficient can have. The default value is `[ ]` (unspecified). Simulink software uses this value to perform:

• Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters”)
• Automatic scaling of fixed-point data types
Programmatic Use
Block Parameter: NumCoeffMax
Type: character vector
Values: scalar
Default: '[]'

Numerator product output — Numerator product output data type
Inherit: Inherit via internal rule (default) | Inherit: Same as input | int8 | int16 | int32 | int64 | fixdt(1,16,0) | <data type expression>

Specify the product output data type for the numerator coefficients. You can set this parameter to:

- A rule that inherits a data type, for example, Inherit: Inherit via internal rule
- A built-in data type, for example, int8
- A data type object, for example, a Simulink.NumericType object
- An expression that evaluates to a data type, for example, fixdt(1,16,0)

Click the Show data type assistant button to display the Data Type Assistant, which helps you set the data type attributes. For more information, see “Specify Data Types Using Data Type Assistant”.

Programmatic Use
Block Parameter: NumProductDataTypeStr
Type: character vector
Values: 'Inherit: Inherit via internal rule' | 'Inherit: Same as input' | 'int8' | 'int16' | 'int32' | 'int64' | 'fixdt(1,16,0)' | '<data type expression>'
Default: 'Inherit: Inherit via internal rule'

Numerator accumulator — Numerator accumulator data type
Inherit: Inherit via internal rule (default) | Inherit: Same as input | Inherit: Same as product output | int8 | int16 | int32 | int64 | fixdt(1,16,0) | <data type expression>

Specify the accumulator data type for the numerator coefficients. You can set this parameter to:

- A rule that inherits a data type, for example, Inherit: Inherit via internal rule
• A built-in data type, for example, int8
• A data type object, for example, a Simulink.NumericType object
• An expression that evaluates to a data type, for example, fixdt(1,16,0)

Click the **Show data type assistant** button to display the **Data Type Assistant**, which helps you set the data type attributes. For more information, see “Specify Data Types Using Data Type Assistant”.

**Programmatic Use**

**Block Parameter:** `NumAccumDataTypeStr`  
**Type:** character vector  
**Values:** 'Inherit: Inherit via internal rule' | 'Inherit: Same as input' | 'Inherit: Same as product output' | 'int8' | 'int16' | 'int32' | 'int64' | 'fixdt(1,16)' | 'fixdt(1,16,0)' | '<data type expression>'  
**Default:** 'Inherit: Inherit via internal rule'

**Denominator coefficients — Denominator coefficient data type**

Inherit: Inherit via internal rule (default) | int8 | int16 | int32 | int64 | fixdt(1,16) | fixdt(1,16,0) | '<data type expression>'

Specify the denominator coefficient data type. You can set this parameter to:

• A rule that inherits a data type, for example, Inherit: Inherit via internal rule
• A built-in integer, for example, int8
• A data type object, for example, a Simulink.NumericType object
• An expression that evaluates to a data type, for example, fixdt(1,16,0)

Click the **Show data type assistant** button to display the **Data Type Assistant**, which helps you set the data type attributes. For more information, see “Specify Data Types Using Data Type Assistant”.

**Programmatic Use**

**Block Parameter:** `DenCoeffDataTypeStr`  
**Type:** character vector  
**Values:** 'Inherit: Inherit via internal rule' | 'int8' | 'int16' | 'int32' | 'int64' | 'fixdt(1,16)' | 'fixdt(1,16,0)' | '<data type expression>'
Default: 'Inherit: Inherit via internal rule'

**Denominator coefficient minimum — Minimum value of denominator coefficients**

\[ \text{[] (default) | scalar} \]

Specify the minimum value that a denominator coefficient can have. The default value is \[ \text{[] (unspecified). Simulink software uses this value to perform:} \]

- Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters”)
- Automatic scaling of fixed-point data types

**Programmatic Use**

**Block Parameter:** DenCoeffMin  
**Type:** character vector  
**Values:** scalar  
**Default:** '[]'

**Denominator coefficient maximum — Maximum value of denominator coefficients**

\[ \text{[] (default) | scalar} \]

Specify the maximum value that a denominator coefficient can have. The default value is \[ \text{[] (unspecified). Simulink software uses this value to perform:} \]

- Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters”)
- Automatic scaling of fixed-point data types

**Programmatic Use**

**Block Parameter:** DenCoeffMax  
**Type:** character vector  
**Values:** scalar  
**Default:** '[]'

**Denominator product output — Denominator product output data type**

Inherit: Inherit via internal rule (default) | Inherit: Same as input | int8 | int16 | int32 | int64 | fixdt(1,16,0) | <data type expression>

Specify the product output data type for the denominator coefficients. You can set this parameter to:
• A rule that inherits a data type, for example, Inherit: Inherit via internal rule
• A built-in data type, for example, int8
• A data type object, for example, a Simulink.NumericType object
• An expression that evaluates to a data type, for example, fixdt(1,16,0)

Click the Show data type assistant button to display the Data Type Assistant, which helps you set the data type attributes. For more information, see “Specify Data Types Using Data Type Assistant”.

**Programmatic Use**

**Block Parameter:** DenProductDataTypeStr
**Type:** character vector
**Values:** 'Inherit: Inherit via internal rule' | 'Inherit: Same as input' | 'int8' | 'int16' | 'int32' | 'int64' | 'fixdt(1,16,0)' | '<data type expression>'
**Default:** 'Inherit: Inherit via internal rule'

**Denominator accumulator — Denominator accumulator data type**

**Inherit:** Inherit via internal rule (default) | Inherit: Same as input | Inherit: Same as product output | int8 | int16 | int32 | int64 | fixdt(1,16,0) | <data type expression>

Specify the accumulator data type for the denominator coefficients. You can set this parameter to:

• A rule that inherits a data type, for example, Inherit: Inherit via internal rule
• A built-in data type, for example, int8
• A data type object, for example, a Simulink.NumericType object
• An expression that evaluates to a data type, for example, fixdt(1,16,0)

Click the Show data type assistant button to display the Data Type Assistant, which helps you set the data type attributes. For more information, see “Specify Data Types Using Data Type Assistant”.

**Programmatic Use**

**Block Parameter:** DenAccumDataTypeStr
**Type:** character vector  
**Values:** 'Inherit: Inherit via internal rule' | 'Inherit: Same as input' | 'Inherit: Same as product output' | 'int8' | 'int16' | 'int32' | 'int64' | 'fixdt(1,16)' | 'fixdt(1,16,0)' | '<data type expression>'  
**Default:** 'Inherit: Inherit via internal rule'

**Output — Output data type**

Inherit: Inherit via internal rule (default) | int8 | int16 | int32 | int64 | fixdt(1,16) | fixdt(1,16,0) | <data type expression>

Specify the output data type. You can set this parameter to:

- A rule that inherits a data type, for example, *Inherit: Inherit via internal rule*
- A built-in data type, for example, int8
- A data type object, for example, a Simulink.NumericType object
- An expression that evaluates to a data type, for example, fixdt(1,16,0)

Click the **Show data type assistant** button to display the **Data Type Assistant**, which helps you set the data type attributes. For more information, see “Specify Data Types Using Data Type Assistant”.

**Programmatic Use**

**Block Parameter:** OutDataTypeStr

**Type:** character vector  
**Values:** 'Inherit: Inherit via internal rule' | 'int8' | 'int16' | 'int32' | 'int64' | 'fixdt(1,16)' | 'fixdt(1,16,0)' | '<data type expression>'  
**Default:** 'Inherit: Inherit via internal rule'

**Output minimum — Minimum value of output**

[ ] (default) | scalar

Specify the minimum value that the block can output. The default value is [ ] (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see “Signal Ranges”)
- Automatic scaling of fixed-point data types
**Programmatic Use**

**Block Parameter:** OutMin  
**Type:** character vector  
**Values:** scalar  
**Default:** '[]'

**Output maximum — Maximum value of output**

[] (default) | scalar

Specify the maximum value that the block can output. The default value is [] (unspecified). Simulink software uses this value to perform:

- Simulation range checking (see “Signal Ranges”)
- Automatic scaling of fixed-point data types

**Programmatic Use**

**Block Parameter:** OutMax  
**Type:** character vector  
**Values:** scalar  
**Default:** '[]'

**Multiplicand data type — Multiplicand data type**

Inherit: Same as input (default) | int8 | int16 | int32 | int64 | fixdt(1,16,0) | <data type expression>

Specify the multiplicand data type. You can set this parameter to:

- A rule that inherits a data type, for example, Inherit: Same as input
- A built-in data type, for example, int8
- A data type object, for example, a Simulink.NumericType object
- An expression that evaluates to a data type, for example, fixdt(1,16,0)

Click the **Show data type assistant** button to display the **Data Type Assistant**, which helps you set the data type attributes. For more information, see “Specify Data Types Using Data Type Assistant”.

**Dependencies**

To enable this parameter, set the **Filter structure** to Direct form I transposed.
**Programmatic Use**

**Block Parameter:** MultiplicandDataTypeStr  
**Type:** character vector  
**Values:** 'Inherit: Same as input' | 'int8' | 'int16' | 'int32' | 'int64' | 'fixdt(1,16,0)' | '<data type expression>'  
**Default:** 'Inherit: Same as input'

**Lock data type settings against changes by the fixed-point tools — Prevent fixed-point tools from overriding data types**  
**off (default) | on**

Select to lock data type settings of this block against changes by the Fixed-Point Tool and the Fixed-Point Advisor. For more information, see “Lock the Output Data Type Setting” (Fixed-Point Designer).

**Programmatic Use**

**Block Parameter:** LockScale  
**Values:** 'off' | 'on'  
**Default:** 'off'

**Integer rounding mode — Rounding mode for fixed-point operations**  
**Floor (default) | Ceiling | Convergent | Nearest | Round | Simplest | Zero**

Specify the rounding mode for fixed-point operations. For more information, see “Rounding” (Fixed-Point Designer).

**Programmatic Use**

**Block Parameter:** RndMeth  
**Type:** character vector  
**Values:** 'Ceiling' | 'Convergent' | 'Floor' | 'Nearest' | 'Round' | 'Simplest' | 'Zero'  
**Default:** 'Floor'

**Saturate on integer overflow — Method of overflow action**  
**off (default) | on**

Specify whether overflows saturate or wrap.
<table>
<thead>
<tr>
<th>Action</th>
<th>Rationale</th>
<th>Impact on Overflows</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Select this check box (on).</td>
<td>Your model has possible overflow, and you want explicit saturation protection in the generated code.</td>
<td>Overflows saturate to either the minimum or maximum value that the data type can represent.</td>
<td>The maximum value that the int8 (signed, 8-bit integer) data type can represent is 127. Any block operation result greater than this maximum value causes overflow of the 8-bit integer. With the check box selected, the block output saturates at 127. Similarly, the block output saturates at a minimum output value of -128.</td>
</tr>
<tr>
<td>Do not select this check box (off).</td>
<td>You want to optimize efficiency of your generated code. You want to avoid overspecifying how a block handles out-of-range signals. For more information, see “Check for Signal Range Errors”.</td>
<td>Overflows wrap to the appropriate value that is representable by the data type.</td>
<td>The maximum value that the int8 (signed, 8-bit integer) data type can represent is 127. Any block operation result greater than this maximum value causes overflow of the 8-bit integer. With the check box cleared, the software interprets the overflow-causing value as int8, which can produce an unintended result. For example, a block result of 130 (binary 1000 0010) expressed as int8, is -126.</td>
</tr>
</tbody>
</table>

When you select this check box, saturation applies to every internal operation on the block, not just the output, or result. Usually, the code generation process can detect when overflow is not possible. In this case, the code generator does not produce saturation code.
**Programmatic Use**

**Block Parameter:** SaturateOnIntegerOverflow  
**Type:** character vector  
**Values:** 'off' | 'on'  
**Default:** 'off'

**State Attributes**

State name — Assign unique name to each state  
' '(default)|'position'|{'a', 'b', 'c'}|a|...

Assign a unique name to each state. If this field is blank (' '), no name assignment occurs.

- To assign a name to a single state, enter the name between quotes, for example, 'position'.
- To assign names to multiple states, enter a comma-delimited list surrounded by braces, for example, {'a', 'b', 'c'}. Each name must be unique.
- To assign state names with a variable in the MATLAB workspace, enter the variable without quotes. A variable can be a character vector, cell array, or structure.

**Limitations**

- The state names apply only to the selected block.
- The number of states must divide evenly among the number of state names.
- You can specify fewer names than states, but you cannot specify more names than states.

For example, you can specify two names in a system with four states. The first name applies to the first two states and the second name to the last two states.

**Dependencies**

To enable this parameter, set **Filter structure** to **Direct form II**.

**Programmatic Use**

**Block Parameter:** StateName  
**Type:** character vector  
**Values:** ' ' | user-defined  
**Default:** ' '
State name must resolve to Simulink signal object — Require state name resolve to a signal object
off (default) | on

Select this check box to require that the state name resolves to a Simulink signal object.

Dependencies

To enable this parameter, set **Filter structure** to **Direct form II** and specify a value for **State name**. This parameter appears only if you set the model configuration parameter **Signal resolution** to a value other than **None**.

Selecting this check box disables **Code generation storage class**.

**Programmatic Use**
**Block Parameter:** StateMustResolveToSignalObject
**Type:** character vector
**Values:** 'off' | 'on'
**Default:** 'off'

Signal object class — Signal object class
Simulink.Signal (default) | object of a class that is derived from Simulink.Signal

Specify the signal object class.

Dependencies

To enable this parameter, set **Filter structure** to **Direct form II** and specify a value for **State name**.

**Programmatic Use**
**Block Parameter:** StateSignalObject
**Type:** character vector
**Values:** object of a class that is derived from Simulink.Signal
**Default:** 'Simulink.Signal'

**Code generation storage class — State storage class for code generation**
Auto (default) | Model default | ExportedGlobal | ImportedExtern |
ImportedExternPointer | BitField (Custom) | Volatile (Custom) |
ExportToFile (Custom) | ImportFromFile (Custom) | FileScope (Custom) |
Localizable (Custom) | Struct (Custom) | GetSet (Custom) | Reusable (Custom)

Select state storage class for code generation.
• **Auto** is the appropriate storage class for states that you do not need to interface to external code.

• **StorageClass** applies the storage class or custom storage class that you select from the list. For information about storage classes, see “Apply Storage Classes to Individual Signal, State, and Parameter Data Elements” (Simulink Coder). For information about custom storage classes, see “Apply Built-In and Customized Storage Classes to Data Elements” (Embedded Coder).

Use **Signal object class** to select custom storage classes from a package other than Simulink.

**Dependencies**

To enable this parameter, set **Filter structure** to **Direct form II**, and specify a value for **State name**.

**Programmatic Use**

**Block Parameter:** StateStorageClass  
**Type:** character vector  
**Values:** 'Auto' | 'Model default' | 'ExportedGlobal' | 'ImportedExtern' | 'ImportedExternPointer' | 'Custom' | ...  
**Default:** 'Auto'

**Code generation storage type qualifier — Storage type qualifier**

' ' (default)

Specify a storage type qualifier such as **const** or **volatile**.

**Note** **TypeQualifier** will be removed in a future release. To apply storage type qualifiers to data, use custom storage classes and memory sections. Unless you use an ERT-based code generation target with Embedded Coder, custom storage classes and memory sections do not affect the generated code.

During simulation, the block uses the following values:

• The initial value of the signal object to which the state name is resolved

• Min and Max values of the signal object

For more information, see “Data Objects”. 

1-484
Dependencies

To enable this parameter, set Filter structure to Direct form II. This parameter is hidden unless you previously set its value.

Programmatic Use
Block Parameter: RTWStateStorageTypeQualifier
Type: character vector
Values: '' | 'const' | 'volatile' | ...
Default: ''

Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>double</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

a. This block only supports signed fixed-point data types.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

PLC Code Generation
Generate Structured Text code using Simulink® PLC Coder™.
Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

This block only supports signed fixed-point data types.

The Discrete Filter block accepts and outputs real and complex signals of any signed numeric data type that Simulink supports. The block supports the same types for the numerator and denominator coefficients.

Numerator and denominator coefficients must have the same complexity. They can have different word lengths and fraction lengths.

The following diagrams show the filter structure and the data types used within the Discrete Filter block for fixed-point signals.

The block omits the dashed divide when you select the Optimize by skipping divide by leading denominator coefficient (a0) parameter.
See Also
Allpole Filter | Digital Filter Design | Discrete FIR Filter | Filter Realization Wizard | dsp.AllpoleFilter | dsp.IIRFilter | filterDesigner | fvtool

Topics
“Sample- and Frame-Based Concepts” (DSP System Toolbox)
“Working with States” on page 13-25

Introduced before R2006a
**Discrete FIR Filter**

Model FIR filters

**Library:**
- Simulink / Discrete
- HDL Coder / Discrete
- HDL Coder / HDL Floating Point Operations

**Description**

The Discrete FIR Filter block independently filters each channel of the input signal with the specified digital FIR filter. The block can implement static filters with fixed coefficients, and time-varying filters with coefficients that change over time. You can tune the coefficients of a static filter during simulation.

This block filters each channel of the input signal independently over time. The **Input processing** parameter allows you to specify whether the block treats each element of the input as an independent channel (sample-based processing), or each column of the input as an independent channel (frame-based processing). To perform frame-based processing, you must have a DSP System Toolbox license.

The output dimensions equal the input dimensions, except when you specify a matrix of filter taps for the **Coefficients** parameter. When you do so, the output dimensions depend on the number of different sets of filter taps you specify.

The outputs of this block numerically match the outputs of the DSP System Toolbox Digital Filter Design block.

This block supports the Simulink state logging feature. For more information, see “State”.

**Filter Structure Support**

You can change the filter structure implemented with the Discrete FIR Filter block by selecting one of the following from the **Filter structure** parameter:

- Direct form
• Direct form symmetric
• Direct form antisymmetric
• Direct form transposed
• Lattice MA

You must have an available DSP System Toolbox license to run a model with any of these filter structures other than Direct form.

### Specifying Initial States

The Discrete FIR Filter block initializes the internal filter states to zero by default, which has the same effect as assuming that past inputs and outputs are zero. You can optionally use the **Initial states** parameter to specify nonzero initial conditions for the filter delays.

To determine the number of initial states you must specify and how to specify them, see the table on valid initial states. The **Initial states** parameter can take one of the forms described in the next table.

#### Valid Initial States

<table>
<thead>
<tr>
<th>Initial Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scalar</td>
<td>The block initializes all delay elements in the filter to the scalar value.</td>
</tr>
<tr>
<td>Vector or matrix (for applying different delay elements to each channel)</td>
<td>Each vector or matrix element specifies a unique initial condition for a corresponding delay element in a corresponding channel:</td>
</tr>
<tr>
<td></td>
<td>• The vector length equals the product of the number of input channels and the number of delay elements in the filter, ( #<em>{\text{of_filter_coeffs}} - 1 ) (or ( #</em>{\text{of_reflection_coeffs}} ) for Lattice MA).</td>
</tr>
<tr>
<td></td>
<td>• The matrix must have the same number of rows as the number of delay elements in the filter, ( #<em>{\text{of_filter_coeffs}} - 1 ) (( #</em>{\text{of_reflection_coeffs}} ) for Lattice MA), and must have one column for each channel of the input signal.</td>
</tr>
</tbody>
</table>
Ports

Input

In — Input signal
scalar | vector | matrix

Input signal to filter, specified as a scalar, vector, or matrix.

Dependencies

When you set Coefficient source to Dialog parameters, the port for the input signal is unlabeled. When you set Coefficient source to Input port, the port for the input signal is labeled In.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

Num — Filter coefficients
scalar | vector | matrix

Specify the filter coefficients as a scalar, vector, or matrix. When you specify a row vector of filter taps, the block applies a single filter to the input. To apply multiple filters to the same input, specify a matrix of coefficients, where each row represents a different set of filter taps.

Dependencies

To enable this port, set Coefficient source to Input port.

To implement multiple filters, Filter structure must be Direct form, and the input must be a scalar.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

External reset — External reset signal
scalar

External reset signal, specified as a scalar. When the specified trigger event occurs, the block resets the states to their initial conditions.
The icon for this port changes based on the value of the **External reset** parameter.

**Dependencies**

To enable this port, set **External reset** to Rising, Falling, Either, Level, or Level hold.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

**— Enable signal**

scalar

Enable signal, specified as a scalar. This port can control execution of the block. The block is enabled when the input to this port is nonzero, and is disabled when the input is 0. The value of the input is checked at the same time step as the block execution.

**Dependencies**

To enable this port, select the **Show enable port** check box.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

**Output**

**Port_1 — Filtered output signal**

scalar | vector | matrix

Filtered output signal.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

**Parameters**

**Main**

**Coefficient source — Source of coefficients**

Dialog parameters (default) | Input port
Choose to specify the filter coefficients using tunable dialog parameters or through an input port, which is useful for time-varying coefficients.

**Programmatic Use**

**Block Parameter:** CoefSource  
**Type:** character vector  
**Values:** 'Dialog parameters' | 'Input port'  
**Default:** 'Dialog parameters'

**Filter structure — Filter structure**

Direct form (default) | Direct form symmetric | Direct form antisymmetric | Direct form transposed | Lattice MA

Select the filter structure you want the block to implement.

**Dependencies**

You must have an available DSP System Toolbox license to run a model with a Discrete FIR Filter block that implements any filter structure other than Direct form.

**Programmatic Use**

**Block Parameter:** FilterStructure  
**Type:** character vector  
**Values:** 'Direct form' | 'Direct form symmetric' | 'Direct form antisymmetric' | 'Direct form transposed' | 'Lattice MA'  
**Default:** 'Direct form'

**Coefficients — Filter coefficients**

[0.5 0.5] (default) | vector | matrix

Specify the coefficient vector for the transfer function. Filter coefficients must be specified as a row vector. When you specify a row vector of filter taps, the block applies a single filter to the input. To apply multiple filters to the same input, specify a matrix of coefficients, where each row represents a different set of filter taps.

**Dependencies**

To enable this parameter, set **Coefficient source** to Dialog parameters.

To implement multiple filters, **Filter structure** must be Direct form, and the input must be a scalar.
Programmatic Use
Block Parameter: Coefficients
Type: character vector
Values: vector
Default: ' [0.5 0.5] '

Input processing — Sample- or frame-based processing
Elements as channels (sample based) (default) | Columns as channels (frame based)

Specify whether the block performs sample- or frame-based processing. You can select one of the following options:

- **Elements as channels (sample based)** — Treat each element of the input as an independent channel (sample-based processing).
- **Columns as channels (frame based)** — Treat each column of the input as an independent channel (frame-based processing).

**Note**: Frame-based processing requires a DSP System Toolbox license.

For more information, see “Sample- and Frame-Based Concepts” (DSP System Toolbox).

Programmatic Use
Block Parameter: InputProcessing
Type: character vector
Values: 'Columns as channels (frame based)' | 'Elements as channels (sample based)'
Default: 'Elements as channels (sample based)'

**Initial states — Initial conditions of filter states**
0 (default) | scalar | vector | matrix

Specify the initial conditions of the filter states. To learn how to specify initial states, see “Specifying Initial States” on page 1-489.

Programmatic Use
Block Parameter: InitialStates
Type: character vector
Values: scalar | vector | matrix
Default: '0'
Show enable port — Create enable port
off (default) | on

Select to control execution of this block with an enable port. The block is considered enabled when the input to this port is nonzero, and is disabled when the input is 0. The value of the input is checked at the same time step as the block execution.

Programmatic Use
Block Parameter: ShowEnablePort
Type: character vector
Values: 'off' | 'on'
Default: 'off'

External reset — External state reset
None (default) | Rising | Falling | Either | Level | Level hold

Specify the trigger event to use to reset the states to the initial conditions.

<table>
<thead>
<tr>
<th>Reset Mode</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>No reset</td>
</tr>
<tr>
<td>Rising</td>
<td>Reset on a rising edge</td>
</tr>
<tr>
<td>Falling</td>
<td>Reset on a falling edge</td>
</tr>
<tr>
<td>Either</td>
<td>Reset on either a rising or falling edge</td>
</tr>
<tr>
<td>Level</td>
<td>Reset in either of these cases:</td>
</tr>
<tr>
<td></td>
<td>• When the reset signal is nonzero at the current time step</td>
</tr>
<tr>
<td></td>
<td>• When the reset signal value changes from nonzero at the previous time step to zero at the current time step</td>
</tr>
<tr>
<td>Level hold</td>
<td>Reset when the reset signal is nonzero at the current time step</td>
</tr>
</tbody>
</table>

Programmatic Use
Block Parameter: ExternalReset
Type: character vector
Values: 'None' | 'Rising' | 'Falling' | 'Either' | 'Level' | 'Level hold'
Default: 'None'
Sample time (-1 for inherited) — Time interval between samples
-1 (default) | scalar | vector

Specify the time interval between samples. To inherit the sample time, set this parameter to -1. For more information, see “Specify Sample Time”.

Programmatic Use
Block Parameter: SampleTime
Type: character vector
Values: scalar | vector
Default: '-1'

Data Types

Tap sum — Tap sum data type
Inherit: Same as input (default) | Inherit: Inherit via internal rule | int8 | uint8 | int16 | uint16 | int32 | uint32 | int64 | uint64 | fixdt(1,16,0) | <data type expression>

Specify the tap sum data type of a direct form symmetric or direct form antisymmetric filter, which is the data type the filter uses when it sums the inputs prior to multiplication by the coefficients. You can set it to:

- A rule that inherits a data type, for example, Inherit: Inherit via internal rule
- A built-in integer, for example, int8
- A data type object, for example, a Simulink.NumericType object
- An expression that evaluates to a data type, for example, fixdt(1,16,0)

Click the Show data type assistant button to display the Data Type Assistant, which helps you set the data type attributes. For more information, see “Specify Data Types Using Data Type Assistant”.

Dependencies

This parameter is only visible when you set the Filter structure to Direct form symmetric or Direct form antisymmetric.

Programmatic Use
Block Parameter: TapSumDataTypeStr
**Type:** character vector

**Values:** 'Inherit: Same as input' | 'int8' | 'uint8' | 'int16' | 'uint16' | 'int32' | 'uint32' | 'int64' | 'uint64' | 'fixdt(1,16,0)' | '<data type expression>'

**Default:** 'Inherit: Same as input'

**Coefficients** — **Coefficient data type**

Inherit: Same word length as input (default) | int8 | uint8 | int16 | uint16 | int32 | uint32 | int64 | uint64 | fixdt(1,16,0) | <data type expression>

Specify the coefficient data type. You can set it to:

- A rule that inherits a data type, for example, Inherit: Same word length as input
- A built-in integer, for example, int8
- A data type object, for example, a Simulink.NumericType object
- An expression that evaluates to a data type, for example, fixdt(1,16,0)

Click the **Show data type assistant** button to display the **Data Type Assistant**, which helps you set the data type attributes. For more information, see “Specify Data Types Using Data Type Assistant”.

**Programmatic Use**

**Block Parameter:** CoeffDataTypeStr

**Type:** character vector

**Values:** 'Inherit: Same word length as input' | 'int8' | 'uint8' | 'int16' | 'uint16' | 'int32' | 'uint32' | 'int64' | 'uint64' | 'fixdt(1,16)' | 'fixdt(1,16,0)' | '<data type expression>'

**Default:** 'Inherit: Same word length as input'

**Coefficients minimum** — **Minimum value of coefficients**

[] (default) | scalar

Specify the minimum value that a filter coefficient should have. The default value is [] (unspecified). Simulink software uses this value to perform:

- Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters”)
- Automatic scaling of fixed-point data types
Programmatic Use
Block Parameter: CoeffMin
Type: character vector
Values: scalar
Default: '[]'

Coefficients maximum — Maximum value of coefficients
[] (default) | scalar

Specify the maximum value that a filter coefficient should have. The default value is [] (unspecified). Simulink software uses this value to perform:

- Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters”)
- Automatic scaling of fixed-point data types

Programmatic Use
Block Parameter: CoeffMax
Type: character vector
Values: scalar
Default: '[]'

Product output — Product output data type
Inherit: Inherit via internal rule (default) | Inherit: Same as input | int8 | uint8 | int16 | uint16 | int32 | uint32 | int64 | uint64 | fixdt(1,16,0) | <data type expression>

Specify the product output data type. You can set it to:

- A rule that inherits a data type, for example, Inherit: Inherit via internal rule
- A built-in data type, for example, int8
- A data type object, for example, a Simulink.NumericType object
- An expression that evaluates to a data type, for example, fixdt(1,16,0)

Click the Show data type assistant button to display the Data Type Assistant, which helps you set the data type attributes. For more information, see “Specify Data Types Using Data Type Assistant”.
### Programmatic Use

**Block Parameter:** ProductDataTypeStr  
**Type:** character vector  
**Values:** 'Inherit: Inherit via internal rule' | 'Inherit: Same as input' | 'int8' | 'uint8' | 'int16' | 'uint16' | 'int32' | 'uint32' | 'int64' | 'uint64' | 'fixdt(1,16,0)' | '<data type expression>'  
**Default:** 'Inherit: Inherit via internal rule'

### Accumulator — Accumulator data type

**Inherit:** Inherit via internal rule (default) | Inherit: Same as input | Inherit: Same as product output | int8 | uint8 | int16 | uint16 | int32 | uint32 | int64 | uint64 | fixdt(1,16,0) | '<data type expression>'

Specify the accumulator data type. You can set it to:

- A rule that inherits a data type, for example, **Inherit: Inherit via internal rule**
- A built-in data type, for example, int8
- A data type object, for example, a Simulink.NumericType object
- An expression that evaluates to a data type, for example, fixdt(1,16,0)

Click the **Show data type assistant** button to display the **Data Type Assistant**, which helps you set the data type attributes. For more information, see “Specify Data Types Using Data Type Assistant”.

### Programmatic Use

**Block Parameter:** AccumDataTypeStr  
**Type:** character vector  
**Values:** 'Inherit: Inherit via internal rule' | 'Inherit: Same as input' | 'Inherit: Same as product output' | 'int8' | 'uint8' | 'int16' | 'uint16' | 'int32' | 'uint32' | 'int64' | 'uint64' | 'fixdt(1,16,0)' | '<data type expression>'  
**Default:** 'Inherit: Inherit via internal rule'

### State — State data type

**Inherit:** Same as accumulator (default) | Inherit: Same as input | int8 | uint8 | int16 | uint16 | int32 | uint32 | int64 | uint64 | fixdt(1,16,0) | '<data type expression>'

Specify the state data type. You can set it to:
• A rule that inherits a data type, for example, Inherit: Same as accumulator
• A built-in integer, for example, int8
• A data type object, for example, a Simulink.NumericType object
• An expression that evaluates to a data type, for example, fixdt(1,16,0)

Dependencies
To enable this parameter, set the Filter structure to Lattice MA.

Programmatic Use
Block Parameter: StateDataTypeStr
Type: character vector
Values: 'Inherit: Same as accumulator' | 'Inherit: Same as input' | int8 | uint8 | int16 | uint16 | int32 | uint32 | int64 | uint64 | fixdt(1,16) | fixdt(1,16,0) | <data type expression>
Default: 'Inherit: Same as accumulator'

Output — Output data type
Inherit: Same as accumulator (default) | Inherit: Same as input | int8 | uint8 | int16 | uint16 | int32 | uint32 | int64 | uint64 | fixdt(1,16) | fixdt(1,16,0) | <data type expression>

Specify the output data type. You can set it to:
• A rule that inherits a data type, for example, Inherit: Same as accumulator
• A built-in data type, for example, int8
• A data type object, for example, a Simulink.NumericType object
• An expression that evaluates to a data type, for example, fixdt(1,16,0)

Click the Show data type assistant button to display the Data Type Assistant, which helps you set the data type attributes. For more information, see “Specify Data Types Using Data Type Assistant”.

Programmatic Use
Block Parameter: OutDataTypeStr
Type: character vector
Values: 'Inherit: Same as accumulator' | 'Inherit: Same as input' | int8 | uint8 | int16 | uint16 | int32 | uint32 | int64 | uint64 | fixdt(1,16) | fixdt(1,16,0) | <data type expression>
| 'uint64' | 'fixdt(1,16)' | 'fixdt(1,16,0)' | '<data type expression>'

**Default:** 'Inherit: Same as accumulator'

**Output minimum — Minimum output value for range checking**

[] (default) | scalar

Lower value of the output range that Simulink checks.

Simulink uses the minimum to perform:

- Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters”) for some blocks.
- Simulation range checking (see “Signal Ranges” and “Enable Simulation Range Checking”).
- Automatic scaling of fixed-point data types.
- Optimization of the code that you generate from the model. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. For more information, see “Optimize using the specified minimum and maximum values” (Simulink Coder).

**Note** Output minimum does not saturate or clip the actual output signal. Use the Saturation block instead.

**Programmatic Use**
**Block Parameter:** OutMin
**Type:** character vector
**Values:** '[ ]' | scalar
**Default:** '[ ]'

**Output maximum — Maximum output value for range checking**

[] (default) | scalar

Upper value of the output range that Simulink checks.

Simulink uses the maximum value to perform:

- Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters”) for some blocks.
• Simulation range checking (see “Signal Ranges” and “Enable Simulation Range Checking”).
• Automatic scaling of fixed-point data types.
• Optimization of the code that you generate from the model. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. For more information, see “Optimize using the specified minimum and maximum values” (Simulink Coder).

**Note**  _Output maximum_ does not saturate or clip the actual output signal. Use the **Saturation** block instead.

**Programmatic Use**
**Block Parameter:** OutMax  
**Type:** character vector  
**Values:** [' []'] | scalar  
**Default:** [' []']

**Lock data type settings against changes by the fixed-point tools — Prevent fixed-point tools from overriding data types**

off (default) | on

Select to lock data type settings of this block against changes by the Fixed-Point Tool and the Fixed-Point Advisor. For more information, see “Lock the Output Data Type Setting” (Fixed-Point Designer).

**Programmatic Use**
**Block Parameter:** LockScale  
**Values:** 'off' | 'on'  
**Default:** 'off'

**Integer rounding mode — Rounding mode for fixed-point operations**

Floor (default) | Ceiling | Convergent | Nearest | Round | Simplest | Zero

Specify the rounding mode for fixed-point operations. For more information, see “Rounding” (Fixed-Point Designer).

**Programmatic Use**
**Block Parameter:** RndMeth  
**Type:** character vector
Values: 'Ceiling' | 'Convergent' | 'Floor' | 'Nearest' | 'Round' | 'Simplest' | 'Zero'
Default: 'Floor'

Saturate on integer overflow — Method of overflow action
off (default) | on

Specify whether overflows saturate or wrap.

<table>
<thead>
<tr>
<th>Action</th>
<th>Rationale</th>
<th>Impact on Overflows</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Select this check box (on).</td>
<td>Your model has possible overflow, and you want explicit saturation protection in the generated code.</td>
<td>Overflows saturate to either the minimum or maximum value that the data type can represent.</td>
<td>The maximum value that the int8 (signed, 8-bit integer) data type can represent is 127. Any block operation result greater than this maximum value causes overflow of the 8-bit integer. With the check box selected, the block output saturates at 127. Similarly, the block output saturates at a minimum output value of -128.</td>
</tr>
<tr>
<td>Action</td>
<td>Rationale</td>
<td>Impact on Overflows</td>
<td>Example</td>
</tr>
<tr>
<td>--------</td>
<td>-----------</td>
<td>---------------------</td>
<td>---------</td>
</tr>
<tr>
<td>Do not select this check box (off).</td>
<td>You want to optimize efficiency of your generated code. You want to avoid overspecifying how a block handles out-of-range signals. For more information, see “Check for Signal Range Errors”.</td>
<td>Overflows wrap to the appropriate value that is representable by the data type.</td>
<td>The maximum value that the int8 (signed, 8-bit integer) data type can represent is 127. Any block operation result greater than this maximum value causes overflow of the 8-bit integer. With the check box cleared, the software interprets the overflow-causing value as int8, which can produce an unintended result. For example, a block result of 130 (binary 1000 0010) expressed as int8, is -126.</td>
</tr>
</tbody>
</table>

When you select this check box, saturation applies to every internal operation on the block, not just the output, or result. Usually, the code generation process can detect when overflow is not possible. In this case, the code generator does not produce saturation code.

**Programmatic Use**

**Block Parameter:** SaturateOnIntegerOverflow  
**Type:** character vector  
**Values:** 'off' | 'on'  
**Default:** 'off'

### Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>double</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

This block supports custom state attributes to customize and generate code more efficiently. To access or set these attributes, in the Simulink editor, select View > Model Data Editor or press Ctrl+Shift+E. For an example, see “Custom State Attributes in Discrete FIR Filter block”.

When Filter structure is set to Direct form, and the signal is real-valued with a data type of single or double, the Discrete FIR Filter block supports SIMD code generation using Intel AVX2 technology. The SIMD technology significantly improves the performance of the generated code, in most cases meeting or exceeding the simulation performance.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

For hardware-friendly valid and reset control signals, and to model exact hardware latency behavior in Simulink, use the Discrete FIR Filter HDL Optimized block instead.

To reduce area or increase speed, the Discrete FIR Filter block supports either block-level optimizations or subsystem-level optimizations. For details of available block-level optimization parameters, see “Block Optimizations” on page 1-507. When you enable block optimizations, the block cannot participate in subsystem optimizations. Use block optimizations when your design is a single one-channel filter. Use subsystem
optimizations to share resources across multiple channels or multiple filters. For the block to participate in subsystem-level optimizations, set the Architecture to Fully parallel. See “Subsystem Optimizations for Filters” (HDL Coder).

**Multichannel Filter Support**

HDL Coder supports the use of vector inputs to Discrete FIR Filter blocks, where each element of the vector represents an independent channel.

1. Connect a vector signal to the Discrete FIR Filter block input port.
2. Specify **Input processing** as **Elements as channels (sample based)**.
3. To reduce area by sharing the filter kernel between channels, set the **ChannelSharing** property to the number of channels.

**Programmable Filter Support**

HDL Coder supports programmable filters for Discrete FIR Filter blocks.

1. On the filter block mask, set **Coefficient source** to **Input port**.
2. Connect a vector signal to the **Num coefficient** port.

Programmable filters are not supported for:

- distributed arithmetic (DA)
- **CoeffMultipliers** set to csd or factored-csd

**Frame-Based Input Support**

HDL Coder supports the use of vector inputs to Discrete FIR Filter blocks, where each element of the vector represents a sample in time. You can use an input vector of up to 512 samples. The frame-based implementation supports fixed-point input and output data types, and uses full-precision internal data types. You can use real input signals with real coefficients, complex input signals with real coefficients, or real input signals with complex coefficients. You can also use frame-based input with programmable coefficients.

1. Connect a vector signal to the Discrete FIR Filter block input port.
2. Specify **Input processing** as **Columns as channels (frame based)**, and set **Filter structure** to **Direct form**.
3. Right-click the block and open **HDL Code > HDL Block Properties**. Set the **Architecture** to **Frame Based**. The block implements a direct form parallel HDL
Frame-based input filters are not supported for:

- Optional block-level reset and enable control signals
- Resettable and enabled subsystems
- Complex input signals with complex coefficients. You can use either complex input signals and real coefficients, or complex coefficients and real input signals.
- Multichannel input
- Sharing and streaming optimizations
- Distributed arithmetic (DA)

**Control Ports**

You can generate HDL code for filters with or without the optional enable port, and with or without the optional reset port.

**Complex Data Support**

You can use any combination of complex input and complex coefficients with fully-parallel filter structures, when you use non-frame-based input data.

Complex coefficients are not supported with serial filter architectures.

When you use frame-based input data you can use either complex input signals and real coefficients, or complex coefficients and real input signals.

You cannot use distributed arithmetic (DA) or **CoeffMultipliers** set to csd or factored-csd with complex coefficients.
## Block Optimizations

<table>
<thead>
<tr>
<th>Area and Speed Optimizations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Serial Architecture</td>
</tr>
<tr>
<td>Distributed Arithmetic</td>
</tr>
<tr>
<td>Multichannel Area Reduction</td>
</tr>
<tr>
<td>Pipelining</td>
</tr>
</tbody>
</table>
## HDL Filter Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>AddPipelineRegisters</strong></td>
<td>Insert a pipeline register between stages of computation in a filter. See also AddPipelineRegisters (HDL Coder).</td>
</tr>
<tr>
<td><strong>ChannelSharing</strong></td>
<td>For a multichannel filter, generate a single filter implementation to be shared between channels. See also ChannelSharing (HDL Coder).</td>
</tr>
<tr>
<td><strong>CoeffMultipliers</strong></td>
<td>Specify the use of canonical signed digit (CSD) optimization to decrease filter area by replacing coefficient multipliers with shift-and-add logic. When you choose a fully parallel filter implementation, you can set CoeffMultipliers to csd or factored-csd. The default is multipliers, which retains multipliers in the HDL. See also CoeffMultipliers (HDL Coder).</td>
</tr>
<tr>
<td><strong>DALUTPartition</strong></td>
<td>Specify distributed arithmetic partial-product LUT partitions as a vector of the sizes of each partition. The sum of all vector elements must be equal to the filter length. The maximum size for a partition is 12 taps. Set DALUTPartition to a scalar value equal to the filter length to generate DA code without LUT partitions. See also DALUTPartition (HDL Coder).</td>
</tr>
<tr>
<td><strong>DARadix</strong></td>
<td>Specify how many distributed arithmetic bit sums are computed in parallel. A DA radix of 8 ($2^3$) generates a DA implementation that computes three sums at a time. The default value is $2^1$, which generates a fully serial DA implementation. See also DARadix (HDL Coder).</td>
</tr>
<tr>
<td><strong>MultiplierInputPipeline</strong></td>
<td>Specify the number of pipeline stages to add at filter multiplier inputs. See also MultiplierInputPipeline (HDL Coder).</td>
</tr>
<tr>
<td><strong>MultiplierOutputPipeline</strong></td>
<td>Specify the number of pipeline stages to add at filter multiplier outputs. See also MultiplierOutputPipeline (HDL Coder).</td>
</tr>
<tr>
<td><strong>ReuseAccum</strong></td>
<td>Enable or disable accumulator reuse in a serial filter implementation. Set ReuseAccum to on to use a cascade-serial implementation. See also ReuseAccum (HDL Coder).</td>
</tr>
<tr>
<td><strong>SerialPartition</strong></td>
<td>Specify partitions for partly serial or cascade-serial filter implementations as a vector of the lengths of each partition. For a fully serial implementation, set this parameter to the length of the filter. See also SerialPartition (HDL Coder).</td>
</tr>
</tbody>
</table>
**HDL Block Properties**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConstrainedOutputPipeline</td>
<td>Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. For more details, see “ConstrainedOutputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td>InputPipeline</td>
<td>Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “InputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td>OutputPipeline</td>
<td>Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “OutputPipeline” (HDL Coder).</td>
</tr>
</tbody>
</table>

**Restrictions**

- HDL code generation is not supported for:
  - Unsigned input data.
  - Nonzero initial states. You must set Initial states to 0.
  - Filter Structure: Lattice MA.
- CoeffMultipliers options are supported only when using a fully parallel architecture. When you select a serial architecture, CoeffMultipliers is hidden from the HDL Block Properties dialog box.

**PLC Code Generation**

Generate Structured Text code using Simulink® PLC Coder™.

**Fixed-Point Conversion**

Design and simulate fixed-point systems using Fixed-Point Designer™.

The Discrete FIR Filter block accepts and outputs real and complex signals of any numeric data type supported by Simulink. The block supports the same types for the coefficients.
The following diagrams show the filter structure and the data types used within the Discrete FIR Filter block for fixed-point signals.

**Direct Form**

You cannot specify the state data type on the block mask for this structure because the input states have the same data types as the input.
**Direct Form Symmetric**

You cannot specify the state data type on the block mask for this structure because the input states have the same data types as the input.

It is assumed that the filter coefficients are symmetric. The block only uses the first half of the coefficients for filtering.
Even Order - Type I

Odd Order - Type II
Even Order - Type I
**Direct Form Antisymmetric**

You cannot specify the state data type on the block mask for this structure because the input states have the same data types as the input.

It is assumed that the filter coefficients are antisymmetric. The block only uses the first half of the coefficients for filtering.
Even Order - Type III

Odd Order - Type IV
Even Order - Type III

Input data type Cast
Tap sum data type

\( b_0 \)

Product output data type Cast
Accumulator data type

\( z^{-1} \)

Product output data type Cast
Accumulator data type

\( b_{M} \)

Product output data type Cast
Accumulator data type

\( z^{-1} \)

Output data type Cast

1

Tap sum data type Cast

1-516
Odd Order - Type IV

**Direct Form Transposed**

States are complex when either the inputs or the coefficients are complex.
1 Blocks — Alphabetical List
Lattice MA
See Also
Digital Filter Design | Discrete Filter

Topics
“Sample- and Frame-Based Concepts” (DSP System Toolbox)
“Custom State Attributes in Discrete FIR Filter block”
“Working with States” on page 13-25

Introduced in R2008a
Discrete PID Controller

Discrete-time or continuous-time PID controller

Library:  
Simulink / Discrete
HDL Coder / Discrete
HDL Coder / HDL Floating Point Operations

Description

The Discrete PID Controller block implements a PID controller (PID, PI, PD, P only, or I only). The block is identical to the PID Controller block with the Time domain parameter set to Discrete-time.

The block output is a weighted sum of the input signal, the integral of the input signal, and the derivative of the input signal. The weights are the proportional, integral, and derivative gain parameters. A first-order pole filters the derivative action.

The block supports several controller types and structures. Configurable options in the block include:

- Controller type (PID, PI, PD, P only, or I only) — See the Controller parameter.
- Controller form (Parallel or Ideal) — See the Form parameter.
- Time domain (continuous or discrete) — See the Time domain parameter.
- Initial conditions and reset trigger — See the Source and External reset parameters.
- Output saturation limits and built-in anti-windup mechanism — See the Limit output parameter.
- Signal tracking for bumpless control transfer and multiloop control — See the Enable tracking mode parameter.

As you change these options, the internal structure of the block changes by activating different variant subsystems. (For more information, see “Variant Subsystems” on page 15-495). To examine the internal structure of the block and its variant subsystems, right-click the block and select Mask > Look Under Mask.
Control Configuration

In one common implementation, the PID Controller block operates in the feedforward path of a feedback loop.

The input of the block is typically an error signal, which is the difference between a reference signal and the system output. For a two-input block that permits setpoint weighting, see Discrete PID Controller (2DOF).

PID Gain Tuning

The PID controller gains are tunable either manually or automatically. Automatic tuning requires Simulink Control Design™ software. For more information about automatic tuning, see the Select tuning method parameter.

Ports

Input

Port_1( u ) — Error signal input
scalar | vector

Difference between a reference signal and the output of the system under control, as shown.
Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

**P — Proportional gain**
scalar | vector

Proportional gain, provided from a source external to the block. External gain input is useful, for example, when you want to map a different PID parameterization to the PID gains of the block. You can also use external gain input to implement gain-scheduled PID control. In gain-scheduled control, you determine the PID coefficients by logic or other calculation in your model and feed them to the block.

**Dependencies**

To enable this port, set **Controller parameters Source** to external.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

**I — Integral gain**
scalar | vector

Integral gain, provided from a source external to the block. External gain input is useful, for example, when you want to map a different PID parameterization to the PID gains of the block. You can also use external gain input to implement gain-scheduled PID control. In gain-scheduled control, you determine the PID coefficients by logic or other calculation in your model and feed them to the block.

When you supply gains externally, time variations in the integral gain are also integrated. This result occurs because of the way the PID gains are implemented within the block. For details, see the **Controller parameters Source** parameter.
Dependencies

To enable this port, set Controller parameters Source to external, and set Controller to a controller type that has integral action.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

D — Derivative gain

scalar | vector

Derivative gain, provided from a source external to the block. External gain input is useful, for example, when you want to map a different PID parameterization to the PID gains of the block. You can also use external gain input to implement gain-scheduled PID control. In gain-scheduled control, you determine the PID coefficients by logic or other calculation in your model and feed them to the block.

When you supply gains externally, time variations in the derivative gain are also differentiated. This result occurs because of the way the PID gains are implemented within the block. For details, see the Controller parameters Source parameter.

Dependencies

To enable this port, set Controller parameters Source to external, and set Controller to a controller type that has derivative action.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

N — Filter coefficient

scalar | vector

Derivative filter coefficient, provided from a source external to the block. External coefficient input is useful, for example, when you want to map a different PID parameterization to the PID gains of the block. You can also use the external input to implement gain-scheduled PID control. In gain-scheduled control, you determine the PID coefficients by logic or other calculation in your model and feed them to the block.

Dependencies

To enable this port, set Controller parameters Source to external, and set Controller to a controller type that has a filtered derivative.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point
**Reset — External reset trigger**

Scalar

Trigger to reset the integrator and filter to their initial conditions. The value of the **External reset** parameter determines whether reset occurs on a rising signal, a falling signal, or a level signal. The port icon indicates the selected trigger type. For example, the following illustration shows a continuous-time PID block with **External reset** set to rising.

![Continuous-time PID block with External reset set to rising](image)

When the trigger occurs, the block resets the integrator and filter to the initial conditions specified by the **Integrator Initial condition** and **Filter Initial condition** parameters or the $I_0$ and $D_0$ ports.

**Note** To be compliant with the Motor Industry Software Reliability Association (MISRA®) software standard, your model must use Boolean signals to drive the external reset ports of the PID controller block.

**Dependencies**

To enable this port, set **External reset** to any value other than none.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point | Boolean

**$I_0$ — Integrator initial condition**

Scalar | Vector

Integrator initial condition, provided from a source external to the block.

**Dependencies**

To enable this port, set **Initial conditions Source** to external, and set **Controller** to a controller type that has integral action.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point
**D_0 — Filter initial condition**
scalar | vector

Initial condition of the derivative filter, provided from a source external to the block.

**Dependencies**

To enable this port, set **Initial conditions Source** to external, and set **Controller** to a controller type that has derivative action.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

**TR — Tracking signal**
scalar | vector

Signal for controller output to track. When signal tracking is active, the difference between the tracking signal and the block output is fed back to the integrator input. Signal tracking is useful for implementing bumpless control transfer in systems that switch between two controllers. It can also be useful to prevent block windup in multiloop control systems. For more information, see the **Enable tracking mode** parameter.

**Dependencies**

To enable this port, select the **Enable tracking mode** parameter.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

**Output**

**Port_1( y ) — Controller output**
scalar | vector

Controller output, generally based on a sum of the input signal, the integral of the input signal, and the derivative of the input signal, weighted by the proportional, integral, and derivative gain parameters. A first-order pole filters the derivative action. Which terms are present in the controller signal depends on what you select for the **Controller** parameter. The base controller transfer function for the current settings is displayed in the **Compensator formula** section of the block parameters and under the mask. Other parameters modify the block output, such as saturation limits specified by the **Upper Limit** and **Lower Limit** saturation parameters.
The controller output is a vector signal when any of the inputs is a vector signal. In that case, the block acts as $N$ independent PID controllers, where $N$ is the number of signals in the input vector.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

## Parameters

**Controller — Controller type**

PID (default) | PI | PD | P | I

Specify which of the proportional, integral, and derivative terms are in the controller.

PID  
Proportional, integral, and derivative action.

PI  
Proportional and integral action only.

PD  
Proportional and derivative action only.

P  
Proportional action only.

I  
Integral action only.

**Tip** The controller transfer function for the current setting is displayed in the Compensator formula section of the block parameters and under the mask.

**Programmatic Use**

**Block Parameter:** Controller

**Type:** string, character vector

**Values:** "PID", "PI", "PD", "P", "I"

**Default:** "PID"

**Form — Controller structure**

Parallel (default) | Ideal
Specify whether the controller structure is parallel or ideal.

Parallel

The controller output is the sum of the proportional, integral, and derivative actions, weighted independently by $P$, $I$, and $D$, respectively. For example, for a continuous-time parallel-form PID controller, the transfer function is:

$$C_{par}(s) = P + I\left(\frac{1}{s}\right) + D\left(\frac{Ns}{s + N}\right).$$

For a discrete-time parallel-form controller, the transfer function is:

$$C_{par}(z) = P + I\alpha(z) + D\left[\frac{N}{1 + N\beta(z)}\right],$$

where the **Integrator method** and **Filter method** parameters determine $\alpha(z)$ and $\beta(z)$, respectively.

Ideal

The proportional gain $P$ acts on the sum of all actions. For example, for a continuous-time ideal-form PID controller, the transfer function is:

$$C_{id}(s) = P\left[1 + I\left(\frac{1}{s}\right) + D\left(\frac{Ns}{s + N}\right)\right].$$

For a discrete-time ideal-form controller, the transfer function is:

$$C_{id}(z) = P\left[1 + I\alpha(z) + D\frac{N}{1 + N\beta(z)}\right],$$

where the **Integrator method** and **Filter method** parameters determine $\alpha(z)$ and $\beta(z)$, respectively.

**Tip** The controller transfer function for the current settings is displayed in the **Compensator formula** section of the block parameters and under the mask.

**Programmatic Use**

**Block Parameter:** Controller

**Type:** string, character vector

**Values:** "Parallel", "Ideal"
Default: "Parallel"

**Time domain — Specify discrete-time or continuous-time controller**
Discrete-time (default) | Continuous-time

When you select Discrete-time, it is recommended that you specify an explicit sample time for the block. See the **Sample time (-1 for inherited)** parameter. Selecting Discrete-time also enables the Integrator method, and Filter method parameters.

When the PID Controller block is in a model with synchronous state control (see the State Control block), you cannot select Continuous-time.

**Note** The PID Controller and Discrete PID Controller blocks are identical except for the default value of this parameter.

**Programmatic Use**

**Block Parameter:** TimeDomain

- **Type:** string, character vector
- **Values:** "Continuous-time", "Discrete-time"
- **Default:** "Continuous-time"

**Sample time (-1 for inherited) — Discrete interval between samples**
-1 (default) | positive scalar

Specify a sample time by entering a positive scalar value, such as 0.1. The default discrete sample time of -1 means that the block inherits its sample time from upstream blocks. However, it is recommended that you set the controller sample time explicitly, especially if you expect the sample time of upstream blocks to change. The effect of the controller coefficients P, I, D, and N depend on the sample time. Thus, for a given set of coefficient values, changing the sample time changes the performance of the controller.

See “Specify Sample Time” for more information.

To implement a continuous-time controller, set **Time domain** to Continuous-time.

**Tip** If you want to run the block with an externally specified or variable sample time, set this parameter to -1 and put the block in a Triggered Subsystem. Then, trigger the subsystem at the desired sample time.
Dependencies

To enable this parameter, set **Time domain** to **Discrete-time**.

**Programmatic Use**

**Block Parameter:** SampleTime  
**Type:** scalar  
**Values:** -1, positive scalar  
**Default:** -1

**Integrator method — Method for computing integral in discrete-time controller**

- **Forward Euler** (default)  
- **Backward Euler**  
- **Trapezoidal**

In discrete time, the integral term of the controller transfer function is $Ia(z)$, where $a(z)$ depends on the integrator method you specify with this parameter.

**Forward Euler**

Forward rectangular (left-hand) approximation,

$$a(z) = \frac{T_s}{z-1}.$$  

This method is best for small sampling times, where the Nyquist limit is large compared to the bandwidth of the controller. For larger sampling times, the **Forward Euler** method can result in instability, even when discretizing a system that is stable in continuous time.

**Backward Euler**

Backward rectangular (right-hand) approximation,

$$a(z) = \frac{T_sz}{z-1}.$$  

An advantage of the **Backward Euler** method is that discretizing a stable continuous-time system using this method always yields a stable discrete-time result.

**Trapezoidal**

Bilinear approximation,

$$a(z) = \frac{T_sz + 1}{2z-1}.$$  

An advantage of the **Trapezoidal** method is that discretizing a stable continuous-time system using this method always yields a stable discrete-time result. Of all

1-530
available integration methods, the Trapezoidal method yields the closest match between frequency-domain properties of the discretized system and the corresponding continuous-time system.

**Tip** The controller formula for the current setting is displayed in the **Compensator formula** section of the block parameters and under the mask.

**Note** For the BackwardEuler or Trapezoidal methods, you cannot generate HDL code for the block if either:

- Limit output is selected and Anti-Windup Method is anything other than none.
- Enable tracking mode is selected.

For more information about discrete-time integration, see the Discrete-Time Integrator block reference page.

**Dependencies**
To enable this parameter, set **Time Domain** to Discrete-time and set **Controller** to a controller type with integral action.

**Programmatic Use**
**Block Parameter:** IntegratorMethod  
Type: string, character vector  
Values: "Forward Euler", "Backward Euler", "Trapezoidal"  
Default: "Forward Euler"

**Filter method — Method for computing derivative in discrete-time controller**  
Forward Euler (default) | Backward Euler | Trapezoidal

In discrete time, the derivative term of the controller transfer function is:

\[
D \left[ \frac{N}{1 + Na(z)} \right]
\]

where \(a(z)\) depends on the filter method you specify with this parameter.

**Forward Euler**  
Forward rectangular (left-hand) approximation,
\[ \alpha(z) = \frac{T_s}{z - 1}. \]

This method is best for small sampling times, where the Nyquist limit is large compared to the bandwidth of the controller. For larger sampling times, the **Forward Euler** method can result in instability, even when discretizing a system that is stable in continuous time.

**Backward Euler**

Backward rectangular (right-hand) approximation,

\[ \alpha(z) = \frac{T_s z}{z - 1}. \]

An advantage of the **Backward Euler** method is that discretizing a stable continuous-time system using this method always yields a stable discrete-time result.

**Trapezoidal**

Bilinear approximation,

\[ \alpha(z) = \frac{T_s z + 1}{2 z - 1}. \]

An advantage of the **Trapezoidal** method is that discretizing a stable continuous-time system using this method always yields a stable discrete-time result. Of all available integration methods, the **Trapezoidal** method yields the closest match between frequency-domain properties of the discretized system and the corresponding continuous-time system.

**Tip** The controller formula for the current setting is displayed in the **Compensator formula** section of the block parameters and under the mask.

For more information about discrete-time integration, see the Discrete-Time Integrator block reference page.

**Dependencies**

To enable this parameter, set **Time Domain** to **Discrete-time** and set **Controller** to a controller type with derivative action.
Programmatic Use

Block Parameter: FilterMethod
Type: string, character vector
Values: "Forward Euler", "Backward Euler", "Trapezoidal"
Default: "Forward Euler"

Main

Source — Source for controller gains and filter coefficient
internal (default) | external

Enabling external inputs for the parameters allows you to compute PID gains and filter coefficients externally to the block and provide them to the block as signal inputs.

internal

Specify the controller gains and filter coefficient using the block parameters P, I, D, and N.

external

Specify the PID gains and filter coefficient externally using block inputs. An additional input port appears on the block for each parameter that is required for the current controller type.

External gain input is useful, for example, when you want to map a different PID parameterization to the PID gains of the block. You can also use external gain input to implement gain-scheduled PID control. In gain-scheduled control, you determine the PID gains by logic or other calculation in your model and feed them to the block.

When you supply gains externally, time variations in the integral and derivative gain values are integrated and differentiated, respectively. This result occurs because in both continuous time and discrete time, the gains are applied to the signal before integration or differentiation. For example, for a continuous-time PID controller with external inputs, the integrator term is implemented as shown in the following illustration.

![Discrete PID Controller](image-url)
Within the block, the input signal $u$ is multiplied by the externally supplied integrator gain, $I$, before integration. This implementation yields:

$$y_i = \int uI \, dt.$$ 

Thus, the integrator gain is included in the integral. Similarly, in the derivative term of the block, multiplication by the derivative gain precedes the differentiation, which causes the derivative gain $D$ to be differentiated.

**Programmatic Use**

**Block Parameter:** ControllerParametersSource

*Type:* string, character vector

*Values:* "internal", "external"

*Default:* "internal"

**Proportional (P) — Proportional gain**

1 (default) | scalar | vector

Specify a finite, real gain value for the proportional gain. When **Controller form** is:

- **Parallel** — Proportional action is independent of the integral and derivative actions. For instance, for a continuous-time parallel PID controller, the transfer function is:

$$C_{par}(s) = P + I\left(\frac{1}{s}\right) + D\left(\frac{Ns}{s + N}\right).$$

For a discrete-time parallel-form controller, the transfer function is:

$$C_{par}(z) = P + I\alpha(z) + D\left[\frac{N}{1 + N\beta(z)}\right],$$

where the **Integrator method** and **Filter method** parameters determine $\alpha(z)$ and $\beta(z)$, respectively.

- **Ideal** — The proportional gain multiplies the integral and derivative terms. For instance, for a continuous-time ideal PID controller, the transfer function is:

$$C_{id}(s) = P\left[1 + I\left(\frac{1}{s}\right) + D\left(\frac{Ns}{s + N}\right)\right].$$

For a discrete-time ideal-form controller, the transfer function is:

$$C_{id}(z) = P\left[1 + I\alpha(z) + D\frac{N}{1 + N\beta(z)}\right].$$
where the **Integrator method** and **Filter method** parameters determine $\alpha(z)$ and $\beta(z)$, respectively.

**Tunable:** Yes

**Dependencies**

To enable this parameter, in the **Main** tab, set the controller-parameters **Source** to internal and set **Controller** to PID, PD, PI, or P.

**Programmatic Use**

**Block Parameter:** P  
**Type:** scalar, vector  
**Default:** 1

**Integral (I) — Integral gain**

1 (default) | scalar | vector

Specify a finite, real gain value for the integral gain.

**Tunable:** Yes

**Dependencies**

To enable this parameter, in the **Main** tab, set the controller-parameters **Source** to internal, and set **Controller** to a type that has integral action.

**Programmatic Use**

**Block Parameter:** I  
**Type:** scalar, vector  
**Default:** 1

**Derivative (D) — Derivative gain**

0 (default) | scalar | vector

Specify a finite, real gain value for the derivative gain.

**Tunable:** Yes

**Dependencies**

To enable this parameter, in the **Main** tab, set the controller-parameters **Source** to internal, and set **Controller** to PID or PD.
Programmatic Use
Block Parameter: D
Type: scalar, vector
Default: 0

Use filtered derivative — Apply filter to derivative term
on (default) | off

For discrete-time PID controllers only, clear this option to replace the filtered derivative with an unfiltered discrete-time differentiator. When you do so, the derivative term of the controller transfer function becomes:

\[ D \frac{z - 1}{zT_s}. \]

For continuous-time PID controllers, the derivative term is always filtered.

Dependencies
To enable this parameter, set Time domain to Discrete-time, and set Controller to a type that has derivative action.

Programmatic Use
Block Parameter: UseFilter
Type: string, character vector
Values: "on", "off"
Default: "on"

Filter coefficient (N) — Derivative filter coefficient
100 (default) | scalar | vector

Specify a finite, real gain value for the filter coefficient. The filter coefficient determines the pole location of the filter in the derivative action of the block. The location of the filter pole depends on the Time domain parameter.

- When Time domain is Continuous-time, the pole location is \( s = -N \).
- When Time domain is Discrete-time, the pole location depends on the Filter method parameter.

<table>
<thead>
<tr>
<th>Filter Method</th>
<th>Location of Filter Pole</th>
</tr>
</thead>
<tbody>
<tr>
<td>Forward Euler</td>
<td>( z_{pole} = 1 - NT_s )</td>
</tr>
</tbody>
</table>
**Filter Method** | **Location of Filter Pole**
--- | ---
Backward Euler | \( z_{pole} = \frac{1}{1 + NT_s} \)
Trapezoidal | \( z_{pole} = \frac{1 - NT_s/2}{1 + NT_s/2} \)

The block does not support \( N = \infty \) (ideal unfiltered derivative). When the **Time domain** is Discrete-time, you can clear **Use filtered derivative** to remove the derivative filter.

**Tunable:** Yes

**Dependencies**

To enable this parameter, in the **Main** tab, set the controller-parameters **Source** to internal and set **Controller** to PID or PD.

**Programmatic Use**

**Block Parameter:** \( N \)

**Type:** scalar, vector

**Default:** 100

**Select tuning method — Tool for automatic tuning of controller coefficients**

- Transfer Function Based (PID Tuner App) (default) | Frequency Response Based

If you have Simulink Control Design software, you can automatically tune the PID coefficients. To do so, use this parameter to select a tuning tool, and click **Tune**.

**Transfer Function Based (PID Tuner App)**

Use **PID Tuner**, which lets you interactively tune PID coefficients while examining relevant system responses to validate performance. By default, **PID Tuner** works with a linearization of your plant model. For models that cannot be linearized, you can tune PID coefficients against a plant model estimated from simulated or measured response data. For more information, see “Introduction to Model-Based PID Tuning in Simulink” (Simulink Control Design).

**Frequency Response Based**

Use **Frequency Response Based PID Tuner**, which tunes PID controller coefficients based on frequency-response estimation data obtained by simulation. This tuning approach is especially useful for plants that are not linearizable or that linearize to...
Both of these tuning methods assume a single-loop control configuration. Simulink Control Design software includes other tuning approaches that suit more complex configurations. For information about other ways to tune a PID Controller block, see “Choose a Control Design Approach” (Simulink Control Design).

**Enable zero-crossing detection** — Detect zero crossings on reset and on entering or leaving a saturation state

| on (default) | off |

Zero-crossing detection can accurately locate signal discontinuities without resorting to excessively small time steps that can lead to lengthy simulation times. If you select **Limit output** or activate **External reset** in your PID Controller block, activating zero-crossing detection can reduce computation time in your simulation. Selecting this parameter activates zero-crossing detection:

- At initial-state reset
- When entering an upper or lower saturation state
- When leaving an upper or lower saturation state

For more information about zero-crossing detection, see “Zero-Crossing Detection”.

**Programmatic Use**

**Block Parameter:** ZeroCross

**Type:** string, character vector

**Values:** "on", "off"

**Default:** "on"

**Initialization**

**Source** — Source for integrator and derivative initial conditions

| internal (default) | external |

Simulink uses initial conditions to initialize the integrator and derivative-filter (or the unfiltered derivative) output at the start of a simulation or at a specified trigger event. (See the **External reset** parameter.) These initial conditions determine the initial block output. Use this parameter to select how to supply the initial condition values to the block.
internal

Specify the initial conditions using the **Integrator Initial condition** and **Filter Initial condition** parameters. If **Use filtered derivative** is not selected, use the **Differentiator** parameter to specify the initial condition for the unfiltered differentiator instead of a filter initial condition.

external

Specify the initial conditions externally using block inputs. Additional input ports $I_o$ and $D_o$ appear on the block. If **Use filtered derivative** is not selected, supply the initial condition for the unfiltered differentiator at $D_o$ instead of a filter initial condition.

**Programmatic Use**

**Block Parameter:** InitialConditionSource
**Type:** string, character vector
**Values:** "internal", "external"
**Default:** "internal"

**Integrator — Integrator initial condition**

0 (default) | scalar | vector

Simulink uses the integrator initial condition to initialize the integrator at the start of a simulation or at a specified trigger event (see **External reset**). The integrator initial condition and the filter initial condition determine the initial output of the PID controller block.

The integrator initial condition cannot be NaN or Inf.

**Dependencies**

To use this parameter, in the **Initialization** tab, set **Source** to internal, and set **Controller** to a type that has integral action.

**Programmatic Use**

**Block Parameter:** InitialConditionForIntegrator
**Type:** scalar, vector
**Default:** 0

**Filter — Filter initial condition**

0 (default) | scalar | vector

Simulink uses the filter initial condition to initialize the derivative filter at the start of a simulation or at a specified trigger event (see **External reset**). The integrator initial
condition and the filter initial condition determine the initial output of the PID controller block.

The filter initial condition cannot be NaN or Inf.

**Dependencies**

To use this parameter, in the **Initialization** tab, set **Source** to internal, and use a controller that has a derivative filter.

**Programmatic Use**

**Block Parameter:** InitialConditionForFilter  
**Type:** scalar, vector  
**Default:** 0

**Differentiator — Initial condition for unfiltered derivative**  
0 (default) | scalar | vector

When you use an unfiltered derivative, Simulink uses this parameter to initialize the differentiator at the start of a simulation or at a specified trigger event (see **External reset**). The integrator initial condition and the derivative initial condition determine the initial output of the PID controller block.

The derivative initial condition cannot be NaN or Inf.

**Dependencies**

To use this parameter, set **Time domain** to Discrete-time, clear the **Use filtered derivative** check box, and in the **Initialization** tab, set **Source** to internal.

**Programmatic Use**

**Block Parameter:** DifferentiatorICPrevScaledInput  
**Type:** scalar, vector  
**Default:** 0

**Initial condition setting — Location at which initial condition is applied**  
State (most efficient) (default) | Output

Use this parameter to specify whether to apply the **Integrator Initial condition** and **Filter Initial condition** parameter to the corresponding block state or output. You can change this parameter at the command line only, using **set_param** to set the **InitialConditionSetting** parameter of the block.
State (most efficient)

Use this option in all situations except when the block is in a triggered subsystem or a function-call subsystem and simplified initialization mode is enabled.

**Output**

Use this option when the block is in a triggered subsystem or a function-call subsystem and simplified initialization mode is enabled.

For more information about the **Initial condition setting** parameter, see the Discrete-Time Integrator block.

This parameter is only accessible through programmatic use.

**Programmatic Use**

**Block Parameter:** InitialConditionSetting  
**Type:** string, character vector  
**Values:** "state", "output"  
**Default:** "state"

**External reset — Trigger for resetting integrator and filter values**

none (default) | rising | falling | either | level

Specify the trigger condition that causes the block to reset the integrator and filter to initial conditions. (If **Use filtered derivative** is not selected, the trigger resets the integrator and differentiator to initial conditions.) Selecting any option other than none enables the **Reset** port on the block for the external reset signal.

**none**

The integrator and filter (or differentiator) outputs are set to initial conditions at the beginning of simulation, and are not reset during simulation.

**rising**

Reset the outputs when the reset signal has a rising edge.

**falling**

Reset the outputs when the reset signal has a falling edge.

**either**

Reset the outputs when the reset signal either rises or falls.

**level**

Reset the outputs when the reset signal either:
• Is nonzero at the current time step
• Changes from nonzero at the previous time step to zero at the current time step

This option holds the outputs to the initial conditions while the reset signal is nonzero.

**Dependencies**

To enable this parameter, set **Controller** to a type that has derivative or integral action.

**Programmatic Use**

**Block Parameter:** ExternalReset  
**Type:** string, character vector  
**Values:** "none","rising","falling","either","level"  
**Default:** "none"

**Ignore reset when linearizing — Force linearization to ignore reset**

**off** (default) | **on**

Select to force Simulink and Simulink Control Design linearization commands to ignore any reset mechanism specified in the **External reset** parameter. Ignoring reset states allows you to linearize a model around an operating point even if that operating point causes the block to reset.

**Programmatic Use**

**Block Parameter:** IgnoreLimit  
**Type:** string, character vector  
**Values:** "off","on"  
**Default:** "off"

**Enable tracking mode — Activate signal tracking**

**off** (default) | **on**

Signal tracking lets the block output follow a tracking signal that you provide at the TR port. When signal tracking is active, the difference between the tracking signal and the block output is fed back to the integrator input with a gain Kt, specified by the **Tracking gain (Kt)** parameter. Signal tracking has several applications, including bumpless control transfer and avoiding windup in multiloop control structures.

**Bumpless control transfer**

Use signal tracking to achieve bumpless control transfer in systems that switch between two controllers. Suppose you want to transfer control between a PID controller and
another controller. To do so, connecting the controller output to the TR input as shown in the following illustration.

For more information, see “Bumpless Control Transfer” on page 14-128.

**Multiloop control**

Use signal tracking to prevent block windup in multiloop control approaches, as in the following model.

The Inner Loop subsystem contains the blocks shown in the following diagram.
Because the PID controller tracks the output of the inner loop, its output never exceeds the saturated inner-loop output. For more details, see “Prevent Block Windup in Multiloop Control” on page 14-127.

Dependencies

To enable this parameter, set Controller to a type that has integral action.

Programmatic Use

Block Parameter: TrackingMode
Type: string, character vector
Values: "off", "on"
Default: "off"

Tracking coefficient (Kt) — Gain of signal-tracking feedback loop
1 (default) | scalar

When you select Enable tracking mode, the difference between the signal TR and the block output is fed back to the integrator input with a gain Kt. Use this parameter to specify the gain in that feedback loop.

Dependencies

To enable this parameter, select Enable tracking mode.

Programmatic Use

Block Parameter: Kt
Type: scalar
Default: 1

Output saturation

Limit Output — Limit block output to specified saturation values
off (default) | on
Activating this option limits the block output internally to the block, so that you do not need a separate Saturation on page 1-1704 block after the controller. It also allows you to activate the anti-windup mechanism built into the block (see the Anti-windup method parameter). Specify the saturation limits using the Lower saturation limit and Upper saturation limit parameters.

**Programmatic Use**

**Block Parameter:** LimitOutput  
**Type:** string, character vector  
**Values:** "off", "on"  
**Default:** "off"

**Upper limit — Upper saturation limit for block output**

Inf (default) | scalar

Specify the upper limit for the block output. The block output is held at the Upper saturation limit whenever the weighted sum of the proportional, integral, and derivative actions exceeds that value.

**Dependencies**

To enable this parameter, select Limit output.

**Programmatic Use**

**Block Parameter:** UpperSaturationLimit  
**Type:** scalar  
**Default:** Inf

**Lower limit — Lower saturation limit for block output**

-Inf (default) | scalar

Specify the lower limit for the block output. The block output is held at the Lower saturation limit whenever the weighted sum of the proportional, integral, and derivative actions goes below that value.

**Dependencies**

To enable this parameter, select Limit output.

**Programmatic Use**

**Block Parameter:** LowerSaturationLimit  
**Type:** scalar  
**Default:** -Inf
Ignore saturation when linearizing — Force linearization to ignore output limits
off (default) | on

Force Simulink and Simulink Control Design linearization commands to ignore block output limits specified in the Upper limit and Lower limit parameters. Ignoring output limits allows you to linearize a model around an operating point even if that operating point causes the block to exceed the output limits.

Dependencies

To enable this parameter, select the Limit output parameter.

Programmatic Use
Block Parameter: LinearizeAsGain
Type: string, character vector
Values: "off", "on"
Default: "off"

Anti-windup method — Integrator anti-windup method
none (default) | back-calculation | clamping

When you select Limit output and the weighted sum of the controller components exceeds the specified output limits, the block output holds at the specified limit. However, the integrator output can continue to grow (integrator windup), increasing the difference between the block output and the sum of the block components. In other words, the internal signals in the block can be unbounded even if the output appears bounded by saturation limits. Without a mechanism to prevent integrator windup, two results are possible:

- If the sign of the input signal never changes, the integrator continues to integrate until it overflows. The overflow value is the maximum or minimum value for the data type of the integrator output.
- If the sign of the input signal changes once the weighted sum has grown beyond the output limits, it can take a long time to unwind the integrator and return the weighted sum within the block saturation limit.

In either case, controller performance can suffer. To combat the effects of windup without an anti-windup mechanism, it may be necessary to detune the controller (for example, by reducing the controller gains), resulting in a sluggish controller. To avoid this problem, activate an anti-windup mechanism using this parameter.
none
Do not use an anti-windup mechanism.

back-calculation
Unwind the integrator when the block output saturates by feeding back to the integrator the difference between the saturated and unsaturated control signal. The following diagram represents the back-calculation feedback circuit for a continuous-time controller. To see the actual feedback circuit for your controller configuration, right-click on the block and select Mask > Look Under Mask.

![Back-calculation Feedback Circuit Diagram]

Use the Back-calculation coefficient (Kb) parameter to specify the gain of the anti-windup feedback circuit. It is usually satisfactory to set $K_b = I$, or for controllers with derivative action, $K_b = \sqrt{I \cdot D}$. Back-calculation can be effective for plants with relatively large dead time [1].

clamping
Integration stops when the sum of the block components exceeds the output limits and the integrator output and block input have the same sign. Integration resumes when the sum of the block components exceeds the output limits and the integrator output and block input have opposite sign. Clamping is sometimes referred to as conditional integration.

Clamping can be useful for plants with relatively small dead times, but can yield a poor transient response for large dead times [1].
Dependencies

To enable this parameter, select the Limit output parameter.

Programmatic Use
Block Parameter: AntiWindupMode
Type: string, character vector
Values: "none", "back-calculation","clamping"
Default: "none"

Back-calculation coefficient (Kb) — Gain coefficient of anti-windup feedback loop
1 (default) | scalar

The back-calculation anti-windup method unwinds the integrator when the block output saturates. It does so by feeding back to the integrator the difference between the saturated and unsaturated control signal. Use the Back-calculation coefficient (Kb) parameter to specify the gain of the anti-windup feedback circuit. For more information, see the Anti-windup method parameter.

Dependencies

To enable this parameter, select the Limit output parameter, and set the Anti-windup method parameter to back-calculation.

Programmatic Use
Block Parameter: Kb
Type: scalar
Default: 1

Data Types

The parameters in this tab are primarily of use in fixed-point code generation using Fixed-Point Designer. They define how numeric quantities associated with the block are stored and processed when you generate code.

If you need to configure data types for fixed-point code generation, click Open Fixed-Point Tool and use that tool to configure the rest of the parameters in the tab. For information about using Fixed-Point Tool, see “Autoscaling Data Objects Using the Fixed-Point Tool” (Fixed-Point Designer).
After you use Fixed-Point Tool, you can use the parameters in this tab to make adjustments to fixed-point data-type settings if necessary. For each quantity associated with the block, you can specify:

- Floating-point or fixed-point data type, including whether the data type is inherited from upstream values in the block.
- The minimum and maximum values for the quantity, which determine how the quantity is scaled for fixed-point representation.

For assistance in selecting appropriate values, click to open the Data Type Assistant for the corresponding quantity. For more information, see “Specify Data Types Using Data Type Assistant”.

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Minimum</th>
<th>Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>P product output</td>
<td></td>
<td></td>
</tr>
<tr>
<td>I product output</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D product output</td>
<td></td>
<td></td>
</tr>
<tr>
<td>N product output</td>
<td></td>
<td></td>
</tr>
<tr>
<td>b product output</td>
<td></td>
<td></td>
</tr>
<tr>
<td>c product output</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sum output</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Additional data types
The specific quantities listed in the Data Types tab vary depending on how you configure the PID controller block. In general, you can configure data types for the following types of quantities:

- **Product output** — Stores the result of a multiplication carried out under the block mask. For example, **P product output** stores the output of the gain block that multiplies the block input with the proportional gain **P**.

- **Parameter** — Stores the value of a numeric block parameter, such as **P**, **I**, or **D**.

- **Block output** — Stores the output of a block that resides under the PID controller block mask. For example, use **Integrator output** to specify the data type of the output of the block called Integrator. This block resides under the mask in the Integrator subsystem, and computes integrator term of the controller action.

- **Accumulator** — Stores values associated with a sum block. For example, **SumI2 Accumulator** sets the data type of the accumulator associated with the sum block **SumI2**. This block resides under the mask in the Back Calculation subsystem of the Anti-Windup subsystem.

In general, you can find the block associated with any listed parameter by looking under the PID Controller block mask and examining its subsystems. You can also use the Model Explorer to search under the mask for the listed parameter name, such as **SumI2**. (See “Search and Edit Using Model Explorer”.)

### Matching Input and Internal Data Types

By default, all data types in the block are set to **Inherit: Inherit via internal rule**. With this setting, Simulink chooses data types to balance numerical accuracy, performance, and generated code size, while accounting for the properties of the embedded target hardware.

Under some conditions, incompatibility can occur between data types within the block. For instance, in continuous time, the Integrator block under the mask can accept only signals of type **double**. If the block input signal is a type that cannot be converted to **double**, such as **uint16**, the internal rules for type inheritance generate an error when you generate code.

To avoid such errors, you can use the Data Types settings to force a data type conversion. For instance, you can explicitly set **P product output**, **I product output**, and **D product output** to **double**, ensuring that the signals reaching the continuous-time integrators are of type **double**.
In general, it is not recommended to use the block in continuous time for code generation applications. However, similar data type errors can occur in discrete time, if you explicitly set some values to data types that are incompatible with downstream signal constraints within the block. In such cases, use the Data Types settings to ensure that all data types are internally compatible.

**Fixed-Point Operational Parameters**

**Integer rounding mode — Rounding mode for fixed-point operations**

Floor (default) | Ceiling | Convergent | Nearest | Round | Simplest | Zero

Specify the rounding mode for fixed-point operations. For more information, see “Rounding” (Fixed-Point Designer).

Block parameters always round to the nearest representable value. To control the rounding of a block parameter, enter an expression using a MATLAB rounding function into the mask field.

**Programmatic Use**

**Block Parameter:** RndMeth

**Type:** character vector

**Values:** 'Ceiling' | 'Convergent' | 'Floor' | 'Nearest' | 'Round' | 'Simplest' | 'Zero'

**Default:** 'Floor'

**Saturate on integer overflow — Method of overflow action**

off (default) | on

Specify whether overflows saturate or wrap.

- **off** — Overflows wrap to the appropriate value that the data type can represent.

  For example, the number 130 does not fit in a signed 8-bit integer and wraps to -126.

- **on** — Overflows saturate to either the minimum or maximum value that the data type can represent.

  For example, an overflow associated with a signed 8-bit integer can saturate to -128 or 127.

**Tip**
- Consider selecting this check box when your model has a possible overflow and you want explicit saturation protection in the generated code.
- Consider clearing this check box when you want to optimize efficiency of your generated code.

Clearing this check box also helps you to avoid overspecifying how a block handles out-of-range signals. For more information, see “Check for Signal Range Errors”.
- When you select this check box, saturation applies to every internal operation on the block, not just the output or result.
- In general, the code generation process can detect when overflow is not possible. In this case, the code generator does not produce saturation code.

**Programmatic Use**

**Block Parameter:** SaturateOnIntegerOverflow  
**Type:** character vector  
**Values:** 'off' | 'on'  
**Default:** 'off'

Lock data type settings against changes by the fixed-point tools — Prevent fixed-point tools from overriding data types

off (default) | on

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on this block. For more information, see “Lock the Output Data Type Setting” (Fixed-Point Designer).

**Programmatic Use**

**Block Parameter:** LockScale  
**Type:** character vector  
**Values:** 'off' | 'on'  
**Default:** 'off'

**State Attributes**

The parameters in this tab are primarily of use in code generation.

**State name (e.g., 'position') — Name for continuous-time filter and integrator states**

' ' (default) | character vector
Assign a unique name to the state associated with the integrator or the filter, for continuous-time PID controllers. (For information about state names in a discrete-time PID controller, see the **State name** parameter.) The state name is used, for example:

- For the corresponding variable in generated code
- As part of the storage name when logging states during simulation
- For the corresponding state in a linear model obtain by linearizing the block

A valid state name begins with an alphabetic or underscore character, followed by alphanumeric or underscore characters.

**Dependencies**

To enable this parameter, set **Time domain** to **Continuous-time**.

**Programmatic Use**

Parameter: `IntegratorContinuousStateAttributes`, `FilterContinuousStateAttributes`

Type: character vector

Default: `' '`

**State name — Names for discrete-time filter and integrator states**

empty string (default) | string | character vector

Assign a unique name to the state associated with the integrator or the filter, for discrete-time PID controllers. (For information about state names in a continuous-time PID controller, see the **State name** (e.g., 'position') parameter.)

A valid state name begins with an alphabetic or underscore character, followed by alphanumeric or underscore characters. The state name is used, for example:

- For the corresponding variable in generated code
- As part of the storage name when logging states during simulation
- For the corresponding state in a linear model obtain by linearizing the block

For more information about the use of state names in code generation, see "Apply Storage Classes to Individual Signal, State, and Parameter Data Elements" (Simulink Coder).

**Dependencies**

To enable this parameter, set **Time domain** to **Discrete-time**.
Programmatic Use
Parameter: IntegratorStateIdentifier, FilterStateIdentifier
Type: string, character vector
Default: ""

State name must resolve to Simulink signal object — Require that state name resolve to a signal object
off (default) | on

Select this parameter to require that the discrete-time integrator or filter state name resolves to a Simulink signal object.

Dependencies
To enable this parameter for the discrete-time integrator or filter state:

1. Set Time domain to Discrete-time.
2. Specify a value for the integrator or filter State name.
3. Set the model configuration parameter Signal resolution to a value other than None.

Selecting this check box disables Code generation storage class for the corresponding integrator or filter state.

Programmatic Use
Block Parameter: IntegratorStateMustResolveToSignalObject, FilterStateMustResolveToSignalObject
Type: string, character vector
Values: "off", "on"
Default: "off"

Code generation storage class — Storage class for code generation
Auto (default) | ExportedGlobal | ImportedExtern | ImportedExternPointer

Select state storage class for code generation. If you do not need to interface to external code, select Auto.

For more information, see “Apply Storage Classes to Individual Signal, State, and Parameter Data Elements” (Simulink Coder) and “Apply Built-In and Customized Storage Classes to Data Elements” (Embedded Coder).
Dependencies

To enable this parameter for the discrete-time integrator or filter state:

1. Set **Time domain** to **Discrete-time**.
2. Specify a value for the integrator or filter **State name**.
3. Set the model configuration parameter **Signal resolution** to a value other than **None**.

Programmatic Use

**Block Parameter:** IntegratorRTWStateStorageClass, FilterRTWStateStorageClass  
**Type:** string, character vector  
**Values:** "Auto", "ExportedGlobal", "ImportedExtern" | "ImportedExternPointer"  
**Default:** "Auto"

**Code generation storage type qualifier — Storage type qualifier**  
empty string (default) | character vector | "const" | "volatile" | ...

Specify a storage type qualifier such as **const** or **volatile**.

**Note** This parameter will be removed in a future release. To apply storage type qualifiers to data, use custom storage classes and memory sections. Unless you use an ERT-based code generation target with Embedded Coder, custom storage classes and memory sections do not affect the generated code.

Dependencies

To enable this parameter, set **Code generation storage class** to any value other than **Auto**.

Programmatic Use

**Block Parameter:**  
IntegratorRTWStateStorageTypeQualifier, FilterRTWStateStorageTypeQualifier  
**Type:** string, character vector  
**Values:** "", "const", "volatile"  
**Default:** ""
Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>double</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

References


Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

For continuous-time PID controllers (Time domain set to Continuous-time):

- Consider using “Model Discretizer” to map continuous-time blocks to discrete equivalents that support code generation. To access Model Discretizer, from your model, in the Apps tab, under Control Systems, click Model Discretizer.
- Not recommended for production code.

For discrete-time PID controllers (Time domain set to Discrete-time):

- Depends on absolute time when placed inside a triggered subsystem hierarchy.
- Generated code relies on memcpy or memset functions (string.h) under certain conditions.
HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture
This block has a single, default HDL architecture.

HDL Block Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConstrainedOutputPipeline</td>
<td>Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. For more details, see “ConstrainedOutputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td>InputPipeline</td>
<td>Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “InputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td>OutputPipeline</td>
<td>Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “OutputPipeline” (HDL Coder).</td>
</tr>
</tbody>
</table>

Restrictions

- HDL code generation is supported for discrete-time PID controllers only (Time domain set to Discrete-time).
- If the Integrator method is set to BackwardEuler or Trapezoidal, you cannot generate HDL code for the block under either of the following conditions:
  - Limit output is selected and the Anti-Windup Method is anything other than none.
  - Enable tracking mode is selected.
- HDL code generation does not support these settings:
  - Continuous-time.
• Source > external.
• External reset > rising, falling, either, or level.
• Anti-windup method clamping when you use double inputs.

**PLC Code Generation**
Generate Structured Text code using Simulink® PLC Coder™.

**Fixed-Point Conversion**
Design and simulate fixed-point systems using Fixed-Point Designer™.

Fixed-point code generation is supported for discrete-time PID controllers only (Time domain set to Discrete-time).

**See Also**
Discrete Derivative | Discrete PID Controller (2DOF) | Discrete-Time Integrator | Gain | PID Controller

**Introduced in R2009b**
Discrete PID Controller (2DOF)

Discrete-time or continuous-time two-degree-of-freedom PID controller
Library: Simulink / Discrete

Description

The Discrete PID Controller (2DOF) block implements a two-degree-of-freedom PID controller (PID, PI, or PD). The block is identical to the PID Controller (2DOF) block with the Time domain parameter set to Discrete-time.

The block generates an output signal based on the difference between a reference signal and a measured system output. The block computes a weighted difference signal for the proportional and derivative actions according to the setpoint weights (b and c) that you specify. The block output is the sum of the proportional, integral, and derivative actions on the respective difference signals, where each action is weighted according to the gain parameters P, I, and D. A first-order pole filters the derivative action.

The block supports several controller types and structures. Configurable options in the block include:

- Controller type (PID, PI, or PD) — See the Controller parameter.
- Controller form (Parallel or Ideal) — See the Form parameter.
- Time domain (discrete or continuous) — See the Time domain parameter.
- Initial conditions and reset trigger — See the Source and External reset parameters.
- Output saturation limits and built-in anti-windup mechanism — See the Limit output parameter.
- Signal tracking for bumpless control transfer and multiloop control — See the Enable tracking mode parameter.

As you change these options, the internal structure of the block changes by activating different variant subsystems. (See “Variant Subsystems” on page 15-495.) To examine
the internal structure of the block and its variant subsystems, right-click the block and select Mask > Look Under Mask.

**Control Configuration**

In one common implementation, the PID Controller block operates in the feedforward path of a feedback loop.

For a single-input block that accepts an error signal (a difference between a setpoint and a system output), see Discrete PID Controller.

**PID Gain Tuning**

The PID controller coefficients and the setpoint weights are tunable either manually or automatically. Automatic tuning requires Simulink Control Design software. For more information about automatic tuning, see the Select tuning method parameter.

**Ports**

**Input**

**Ref** — Reference signal

Scalar | Vector

Reference signal for plant to follow, as shown.
When the reference signal is a vector, the block acts separately on each signal, vectorizing the PID coefficients and producing a vector output signal of the same dimensions. You can specify the PID coefficients and some other parameters as vectors of the same dimensions as the input signal. Doing so is equivalent to specifying a separate PID controller for each entry in the input signal.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

**Port_1( y ) — Measured system output**

scalar | vector

Feedback signal for the controller, from the plant output.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

**P — Proportional gain**

scalar | vector

Proportional gain, provided from a source external to the block. External gain input is useful, for example, when you want to map a different PID parameterization to the PID gains of the block. You can also use external gain input to implement gain-scheduled PID control. In gain-scheduled control, you determine the PID coefficients by logic or other calculation in your model and feed them to the block.

**Dependencies**

To enable this port, set **Controller parameters Source** to `external`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`
**I — Integral gain**
scalar | vector

Integral gain, provided from a source external to the block. External gain input is useful, for example, when you want to map a different PID parameterization to the PID gains of the block. You can also use external gain input to implement gain-scheduled PID control. In gain-scheduled control, you determine the PID coefficients by logic or other calculation in your model and feed them to the block.

When you supply gains externally, time variations in the integral gain are also integrated. This result occurs because of the way the PID gains are implemented within the block. For details, see the Controller parameters Source parameter.

**Dependencies**

To enable this port, set Controller parameters Source to external, and set Controller to a controller type that has integral action.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

**D — Derivative gain**
scalar | vector

Derivative gain, provided from a source external to the block. External gain input is useful, for example, when you want to map a different PID parameterization to the PID gains of the block. You can also use external gain input to implement gain-scheduled PID control. In gain-scheduled control, you determine the PID coefficients by logic or other calculation in your model and feed them to the block.

When you supply gains externally, time variations in the derivative gain are also differentiated. This result occurs because of the way the PID gains are implemented within the block. For details, see the Controller parameters Source parameter.

**Dependencies**

To enable this port, set Controller parameters Source to external, and set Controller to a controller type that has derivative action.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

**N — Filter coefficient**
scalar | vector
Derivative filter coefficient, provided from a source external to the block. External coefficient input is useful, for example, when you want to map a different PID parameterization to the PID gains of the block. You can also use the external input to implement gain-scheduled PID control. In gain-scheduled control, you determine the PID coefficients by logic or other calculation in your model and feed them to the block.

**Dependencies**

To enable this port, set **Controller parameters Source** to external, and set **Controller** to a controller type that has a filtered derivative.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

**b — Proportional setpoint weight**

scalar | vector

Proportional setpoint weight, provided from a source external to the block. External input is useful, for example, when you want to map a different PID parameterization to the PID gains of the block. You can also use the external input to implement gain-scheduled PID control. In gain-scheduled control, you determine the PID coefficients by logic or other calculation in your model and feed them to the block.

**Dependencies**

To enable this port, set **Controller parameters Source** to external.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

**c — Derivative setpoint weight**

scalar | vector

Derivative setpoint weight, provided from a source external to the block. External input is useful, for example, when you want to map a different PID parameterization to the PID gains of the block. You can also use the external input to implement gain-scheduled PID control. In gain-scheduled control, you determine the PID coefficients by logic or other calculation in your model and feed them to the block.

**Dependencies**

To enable this port, set **Controller parameters Source** to external, and set **Controller** to a controller type that has derivative action.
Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

Reset — External reset trigger
scalar

Trigger to reset the integrator and filter to their initial conditions. Use the External reset parameter to specify what kind of signal triggers a reset. The port icon indicates the trigger type specified in that parameter. For example, the following illustration shows a continuous-time PID Controller (2DOF) block with External reset set to rising.

When the trigger occurs, the block resets the integrator and filter to the initial conditions specified by the Integrator Initial condition and Filter Initial condition parameters or the $I_0$ and $D_0$ ports.

Note To be compliant with the Motor Industry Software Reliability Association (MISRA) software standard, your model must use Boolean signals to drive the external reset ports of the PID controller block.

Dependencies

To enable this port, set External reset to any value other than none.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point | Boolean

$I_0$ — Integrator initial condition
scalar | vector

Integrator initial condition, provided from a source external to the block.
Dependencies

To enable this port, set Initial conditions Source to external, and set Controller to a controller type that has integral action.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

D₀ — Filter initial condition
scalar | vector

Initial condition of the derivative filter, provided from a source external to the block.

Dependencies

To enable this port, set Initial conditions Source to external, and set Controller to a controller type that has derivative action.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

TR — Tracking signal
scalar | vector

Signal for controller output to track. When signal tracking is active, the difference between the tracking signal and the block output is fed back to the integrator input. Signal tracking is useful for implementing bumpless control transfer in systems that switch between two controllers. It can also be useful to prevent block windup in multiloop control systems. For more information, see the Enable tracking mode parameter.

Dependencies

To enable this port, select the Enable tracking mode parameter.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

Output

Port_1( u ) — Controller output
scalar | vector

Controller output, generally based on a sum of the input signal, the integral of the input signal, and the derivative of the input signal, weighted by the setpoint weights and by the
proportional, integral, and derivative gain parameters. A first-order pole filters the derivative action. Which terms are present in the controller signal depends on what you select for the **Controller** parameter. The base controller transfer function for the current settings is displayed in the **Compensator formula** section of the block parameters and under the mask. Other parameters modify the block output, such as saturation limits specified by the **Upper Limit** and **Lower Limit** saturation parameters.

The controller output is a vector signal when any of the inputs is a vector signal. In that case, the block acts as \( N \) independent PID controllers, where \( N \) is the number of signals in the input vector.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

## Parameters

**Controller — Controller type**

PID (default) | PI | PD

Specify which of the proportional, integral, and derivative terms are in the controller.

**PID**

Proportional, integral, and derivative action.

**PI**

Proportional and integral action only.

**PD**

Proportional and derivative action only.

**Tip** The controller output for the current setting is displayed in the **Compensator formula** section of the block parameters and under the mask.

### Programmatic Use

**Block Parameter:** Controller  
**Type:** string, character vector  
**Values:** "PID", "PI", "PD"  
**Default:** "PID"
Form — Controller structure
Parallel (default) | Ideal

Specify whether the controller structure is parallel or ideal.

Parallel
The proportional, integral, and derivative gains $P$, $I$, and $D$, are applied independently. For example, for a continuous-time 2-DOF PID controller in parallel form, the controller output $u$ is:

$$u = P(br - y) + I \frac{1}{s}(r - y) + D \frac{N}{1 + N} (cr - y),$$

where $r$ is the reference signal, $y$ is the measured plant output signal, and $b$ and $c$ are the setpoint weights.

For a discrete-time 2-DOF controller in parallel form, the controller output is:

$$u = P(br - y) + Ia(z)(r - y) + D \frac{N}{1 + N}(cr - y),$$

where the Integrator method and Filter method parameters determine $a(z)$ and $\beta(z)$, respectively.

Ideal
The proportional gain $P$ acts on the sum of all actions. For example, for a continuous-time 2-DOF PID controller in ideal form, the controller output is:

$$u = P \left[ (br - y) + I \frac{1}{s}(r - y) + D \frac{N}{1 + N} (cr - y) \right].$$

For a discrete-time 2-DOF PID controller in ideal form, the transfer function is:

$$u = P \left[ (br - y) + Ia(z)(r - y) + D \frac{N}{1 + N}(cr - y) \right],$$

where the Integrator method and Filter method parameters determine $a(z)$ and $\beta(z)$, respectively.

Tip The controller output for the current settings is displayed in the Compensator formula section of the block parameters and under the mask.
Programmatic Use
Block Parameter: Controller
Type: string, character vector
Values: "Parallel", "Ideal"
Default: "Parallel"

Time domain — Specify discrete-time or continuous-time controller
Discrete-time (default) | Continuous-time

When you select Discrete-time, it is recommended that you specify an explicit sample time for the block. See the Sample time (-1 for inherited) parameter. Selecting Discrete-time also enables the Integrator method, and Filter method parameters.

When the PID Controller block is in a model with synchronous state control (see the State Control block), you cannot select Continuous-time.

Note The PID Controller (2DOF) and Discrete PID Controller (2DOF) blocks are identical except for the default value of this parameter.

Programmatic Use
Block Parameter: TimeDomain
Type: string, character vector
Values: "Continuous-time", "Discrete-time"
Default: "Discrete-time"

Sample time (-1 for inherited) — Discrete interval between samples
-1 (default) | positive scalar

Specify a sample time by entering a positive scalar value, such as 0.1. The default discrete sample time of -1 means that the block inherits its sample time from upstream blocks. However, it is recommended that you set the controller sample time explicitly, especially if you expect the sample time of upstream blocks to change. The effect of the controller coefficients P, I, D, and N depend on the sample time. Thus, for a given set of coefficient values, changing the sample time changes the performance of the controller.

See “Specify Sample Time” for more information.

To implement a continuous-time controller, set Time domain to Continuous-time.
Tip If you want to run the block with an externally specified or variable sample time, set this parameter to -1 and put the block in a Triggered Subsystem. Then, trigger the subsystem at the desired sample time.

Dependencies

To enable this parameter, set **Time domain** to **Discrete-time**.

Programmatic Use

**Block Parameter:** SampleTime  
**Type:** scalar  
**Values:** -1, positive scalar  
**Default:** -1

**Integrator method — Method for computing integral in discrete-time controller**  
*Forward Euler* (default) | *Backward Euler* | *Trapezoidal*

In discrete time, the integral term of the controller transfer function is \( I_a(z) \), where \( a(z) \) depends on the integrator method you specify with this parameter.

**Forward Euler**

Forward rectangular (left-hand) approximation,

\[
\alpha(z) = \frac{T_s}{z - 1}.
\]

This method is best for small sampling times, where the Nyquist limit is large compared to the bandwidth of the controller. For larger sampling times, the **Forward Euler** method can result in instability, even when discretizing a system that is stable in continuous time.

**Backward Euler**

Backward rectangular (right-hand) approximation,

\[
\alpha(z) = \frac{T_s z}{z - 1}.
\]

An advantage of the **Backward Euler** method is that discretizing a stable continuous-time system using this method always yields a stable discrete-time result.

**Trapezoidal**

Bilinear approximation,
\[ \alpha(z) = \frac{T_s z + 1}{2z - 1}. \]

An advantage of the Trapezoidal method is that discretizing a stable continuous-time system using this method always yields a stable discrete-time result. Of all available integration methods, the Trapezoidal method yields the closest match between frequency-domain properties of the discretized system and the corresponding continuous-time system.

**Tip** The controller formula for the current setting is displayed in the **Compensator formula** section of the block parameters and under the mask.

For more information about discrete-time integration, see the Discrete-Time Integrator block reference page.

**Dependencies**

To enable this parameter, set **Time Domain** to **Discrete-time** and set **Controller** to a controller type with integral action.

**Programmatic Use**

**Block Parameter:** IntegratorMethod  
**Type:** string, character vector  
**Values:** "Forward Euler", "Backward Euler", "Trapezoidal"  
**Default:** "Forward Euler"

**Filter method — Method for computing derivative in discrete-time controller**

Forward Euler (default) | Backward Euler | Trapezoidal

In discrete time, the derivative term of the controller transfer function is:

\[ D \left[ \frac{N}{1 + Na(z)} \right], \]

where \( \alpha(z) \) depends on the filter method you specify with this parameter.

**Forward Euler**

Forward rectangular (left-hand) approximation,

\[ \alpha(z) = \frac{T_s}{z - 1}. \]
This method is best for small sampling times, where the Nyquist limit is large compared to the bandwidth of the controller. For larger sampling times, the Forward Euler method can result in instability, even when discretizing a system that is stable in continuous time.

**Backward Euler**

Backward rectangular (right-hand) approximation,

\[
\alpha(z) = \frac{T_s z}{z - 1}.
\]

An advantage of the Backward Euler method is that discretizing a stable continuous-time system using this method always yields a stable discrete-time result.

**Trapezoidal**

Bilinear approximation,

\[
\alpha(z) = \frac{T_s z + 1}{2 z - 1}.
\]

An advantage of the Trapezoidal method is that discretizing a stable continuous-time system using this method always yields a stable discrete-time result. Of all available integration methods, the Trapezoidal method yields the closest match between frequency-domain properties of the discretized system and the corresponding continuous-time system.

**Tip** The controller formula for the current setting is displayed in the **Compensator formula** section of the block parameters and under the mask.

For more information about discrete-time integration, see the Discrete-Time Integrator block reference page.

**Dependencies**

To enable this parameter, set **Time Domain** to Discrete-time and set **Controller** to a controller type with derivative action.

**Programmatic Use**

**Block Parameter:** FilterMethod  
**Type:** string, character vector  
**Values:** "Forward Euler", "Backward Euler", "Trapezoidal"
Default: "Forward Euler"

Main

Source — Source for controller gains and filter coefficient
internal (default) | external

internal
Specify the controller gains, filter coefficient, and setpoint weights using the block parameters P, I, D, N, b, and c respectively.

external
Specify the PID gains, filter coefficient, and setpoint weights externally using block inputs. An additional input port appears on the block for each parameter that is required for the current controller type.

Enabling external inputs for the parameters allows you to compute their values externally to the block and provide them to the block as signal inputs.

External input is useful, for example, when you want to map a different PID parameterization to the PID gains of the block. You can also use external gain input to implement gain-scheduled PID control. In gain-scheduled control, you determine the PID gains by logic or other calculation in your model and feed them to the block.

When you supply gains externally, time variations in the integral and derivative gain values are integrated and differentiated, respectively. The derivative setpoint weight c is also differentiated. This result occurs because in both continuous time and discrete time, the gains are applied to the signal before integration or differentiation. For example, for a continuous-time PID controller with external inputs, the integrator term is implemented as shown in the following illustration.

Within the block, the input signal u is multiplied by the externally supplied integrator gain, I, before integration. This implementation yields:

\[ u_i = \int (r - y)I \, dt. \]
Thus, the integrator gain is included in the integral. Similarly, in the derivative term of the block, multiplication by the derivative gain precedes the differentiation, which causes the derivative gain $D$ and the derivative setpoint weight $c$ to be differentiated.

**Programmatic Use**

**Block Parameter:** ControllerParametersSource  
**Type:** string, character vector  
**Values:** "internal", "external"  
**Default:** "internal"

**Proportional (P) — Proportional gain**

1 (default) | scalar | vector

Specify a finite, real gain value for the proportional gain. When **Controller form** is:

- **Parallel** — Proportional action is independent of the integral and derivative actions. For example, for a continuous-time 2-DOF PID controller in parallel form, the controller output $u$ is:

$$ u = P(br - y) + I\frac{1}{s}(r - y) + D\frac{N}{1 + N\frac{1}{s}}(cr - y), $$

where $r$ is the reference signal, $y$ is the measured plant output signal, and $b$ and $c$ are the setpoint weights.

For a discrete-time 2-DOF controller in parallel form, the controller output is:

$$ u = P(br - y) + I\alpha(z)(r - y) + D\frac{N}{1 + N\beta(z)}(cr - y), $$

where the **Integrator method** and **Filter method** parameters determine $\alpha(z)$ and $\beta(z)$, respectively.

- **Ideal** — The proportional gain multiplies the integral and derivative terms. For example, for a continuous-time 2-DOF PID controller in ideal form, the controller output is:

$$ u = P\left((br - y) + I\frac{1}{s}(r - y) + D\frac{N}{1 + N\frac{1}{s}}(cr - y)\right). $$

For a discrete-time 2-DOF PID controller in ideal form, the transfer function is:
\[ u = P \left[ (br - y) + I\alpha(z)(r - y) + D \frac{N}{1 + N\beta(z)}(cr - y) \right], \]

where the **Integrator method** and **Filter method** parameters determine \( \alpha(z) \) and \( \beta(z) \), respectively.

**Tunable:** Yes

**Dependencies**

To enable this parameter, set the Controller parameters **Source** to **internal**.

**Programmatic Use**

**Block Parameter:** \( P \)

**Type:** scalar, vector

**Default:** 1

**Integral (I) — Integral gain**

1 (default) | scalar | vector

Specify a finite, real gain value for the integral gain.

**Tunable:** Yes

**Dependencies**

To enable this parameter, in the **Main** tab, set the controller-parameters **Source** to **internal**, and set **Controller** to a type that has integral action.

**Programmatic Use**

**Block Parameter:** \( I \)

**Type:** scalar, vector

**Default:** 1

**Derivative (D) — Derivative gain**

0 (default) | scalar | vector

Specify a finite, real gain value for the derivative gain.

**Tunable:** Yes

**Dependencies**

To enable this parameter, in the **Main** tab, set the controller-parameters **Source** to **internal**, and set **Controller** to PID or PD.
**Programmatic Use**

**Block Parameter:** D

**Type:** scalar, vector

**Default:** 0

**Use filtered derivative — Apply filter to derivative term**

*on (default) | off*

For discrete-time PID controllers only, clear this option to replace the filtered derivative with an unfiltered discrete-time differentiator. When you do so, the derivative term of the controller output becomes:

\[
D \frac{z - 1}{zT_s} (cr - y).
\]

For continuous-time PID controllers, the derivative term is always filtered.

**Dependencies**

To enable this parameter, set **Time domain** to **Discrete-time**, and set **Controller** to a type that has a derivative term.

**Programmatic Use**

**Block Parameter:** UseFilter

**Type:** string, character vector

**Values:** "on", "off"

**Default:** "on"

**Filter coefficient (N) — Derivative filter coefficient**

100 (default) | scalar | vector

Specify a finite, real gain value for the filter coefficient. The filter coefficient determines the pole location of the filter in the derivative action of the block. The location of the filter pole depends on the **Time domain** parameter.

- When **Time domain** is **Continuous-time**, the pole location is \( s = -N \).
- When **Time domain** is **Discrete-time**, the pole location depends on the **Filter method** parameter.

<table>
<thead>
<tr>
<th>Filter Method</th>
<th>Location of Filter Pole</th>
</tr>
</thead>
<tbody>
<tr>
<td>Forward Euler</td>
<td>( z_{pole} = 1 - NT_s )</td>
</tr>
</tbody>
</table>
## Filter Method

<table>
<thead>
<tr>
<th>Filter Method</th>
<th>Location of Filter Pole</th>
</tr>
</thead>
<tbody>
<tr>
<td>Backward Euler</td>
<td>$z_{pole} = \frac{1}{1 + NT_s}$</td>
</tr>
<tr>
<td>Trapezoidal</td>
<td>$z_{pole} = \frac{1 - NT_s/2}{1 + NT_s/2}$</td>
</tr>
</tbody>
</table>

The block does not support $N = \infty$ (ideal unfiltered derivative). When the Time domain is Discrete-time, you can clear Use filtered derivative to remove the derivative filter.

**Tunable:** Yes

**Dependencies**

To enable this parameter, in the Main tab, set the controller-parameters Source to internal and set Controller to PID or PD.

**Programmatic Use**

**Block Parameter:** N  
**Type:** scalar, vector  
**Default:** 100

---

**Setpoint weight (b) — Proportional setpoint weight**

1 (default) | scalar | vector

Setpoint weight on the proportional term of the controller. The proportional term of a 2-DOF controller output is $P(br-y)$, where $r$ is the reference signal and $y$ is the measured plant output. Setting $b$ to 0 eliminates proportional action on the reference signal, which can reduce overshoot in the system response to step changes in the setpoint. Changing the relative values of $b$ and $c$ changes the balance between disturbance rejection and setpoint tracking.

**Tunable:** Yes

**Dependencies**

To enable this parameter, in the Main tab, set the controller-parameters Source to internal.

**Programmatic Use**

**Block Parameter:** b
**Type:** scalar, vector

**Default:** 1

**Setpoint weight (c) — Derivative setpoint weight**

1 (default) | scalar | vector

Setpoint weight on the derivative term of the controller. The derivative term of a 2-DOF controller acts on $cr-y$, where $r$ is the reference signal and $y$ is the measured plant output. Thus, setting $c$ to 0 eliminates derivative action on the reference signal, which can reduce transient response to step changes in the setpoint. Setting $c$ to 0 can yield a controller that achieves both effective disturbance rejection and smooth setpoint tracking without excessive transient response. Changing the relative values of $b$ and $c$ changes the balance between disturbance rejection and setpoint tracking.

**Tunable:** Yes

**Dependencies**

To enable this parameter, in the Main tab, set the controller-parameters **Source** to internal and set **Controller** to a type that has derivative action.

**Programmatic Use**

**Block Parameter:** c

**Type:** scalar, vector

**Default:** 1

**Select tuning method — Tool for automatic tuning of controller coefficients**

Transfer Function Based (PID Tuner App) (default) | Frequency Response Based

If you have Simulink Control Design software, you can automatically tune the PID coefficients when they are internal to the block. To do so, use this parameter to select a tuning tool, and click **Tune**.

**Transfer Function Based (PID Tuner App)**

Use **PID Tuner**, which lets you interactively tune PID coefficients while examining relevant system responses to validate performance. **PID Tuner** can tune all the coefficients $P$, $I$, $D$, and $N$, and the setpoint coefficients $b$ and $c$. By default, **PID Tuner** works with a linearization of your plant model. For models that cannot be linearized, you can tune PID coefficients against a plant model estimated from simulated or measured response data. For more information, see “Design Two-Degree-of-Freedom PID Controllers” (Simulink Control Design).
Frequency Response Based

Use **Frequency Response Based PID Tuner**, which tunes PID controller coefficients based on frequency-response estimation data obtained by simulation. This tuning approach is especially useful for plants that are not linearizable or that linearize to zero. **Frequency Response Based PID Tuner** tunes the coefficients $P$, $I$, $D$, and $N$, but does not tune the setpoint coefficients $b$ and $c$. For more information, see “Design PID Controller from Plant Frequency-Response Data” (Simulink Control Design).

Both of these tuning methods assume a single-loop control configuration. Simulink Control Design software includes other tuning approaches that suit more complex configurations. For information about other ways to tune a PID Controller block, see “Choose a Control Design Approach” (Simulink Control Design).

**Dependencies**

To enable this parameter, in the **Main** tab, set the controller-parameters **Source** to **internal**.

**Enable zero-crossing detection — Detect zero crossings on reset and on entering or leaving a saturation state**

*on (default) | off*

Zero-crossing detection can accurately locate signal discontinuities without resorting to excessively small time steps that can lead to lengthy simulation times. If you select **Limit output** or activate **External reset** in your PID Controller block, activating zero-crossing detection can reduce computation time in your simulation. Selecting this parameter activates zero-crossing detection:

- At initial-state reset
- When entering an upper or lower saturation state
- When leaving an upper or lower saturation state

For more information about zero-crossing detection, see “Zero-Crossing Detection”.

**Programmatic Use**

**Block Parameter:** ZeroCross  
**Type:** string, character vector  
**Values:** "on", "off"  
**Default:** "on"
Initialization

Source — Source for integrator and derivative initial conditions

internal (default) | external

Simulink uses initial conditions to initialize the integrator and derivative-filter (or the unfiltered derivative) output at the start of a simulation or at a specified trigger event. (See the External reset parameter.) These initial conditions determine the initial block output. Use this parameter to select how to supply the initial condition values to the block.

internal

Specify the initial conditions using the Integrator Initial condition and Filter Initial condition parameters. If Use filtered derivative is not selected, use the Differentiator parameter to specify the initial condition for the unfiltered differentiator instead of a filter initial condition.

external

Specify the initial conditions externally using block inputs. Additional input ports I_o and D_o appear on the block. If Use filtered derivative is not selected, supply the initial condition for the unfiltered differentiator at D_o instead of a filter initial condition.

Programmatic Use

Block Parameter: InitialConditionSource

Type: string, character vector
Values: "internal", "external"
Default: "internal"

Integrator — Integrator initial condition

0 (default) | scalar | vector

Simulink uses the integrator initial condition to initialize the integrator at the start of a simulation or at a specified trigger event (see External reset). The integrator initial condition and the filter initial condition determine the initial output of the PID controller block.

The integrator initial condition cannot be NaN or Inf.

Dependencies

To use this parameter, in the Initialization tab, set Source to internal, and set Controller to a type that has integral action.
**Programmatic Use**

**Block Parameter:** InitialConditionForIntegrator  
**Type:** scalar, vector  
**Default:** 0

**Filter — Filter initial condition**  
0 (default) | scalar | vector

Simulink uses the filter initial condition to initialize the derivative filter at the start of a simulation or at a specified trigger event (see **External reset**). The integrator initial condition and the filter initial condition determine the initial output of the PID controller block.

The filter initial condition cannot be NaN or Inf.

**Dependencies**

To use this parameter, in the **Initialization** tab, set **Source** to internal, and use a controller that has a derivative filter.

**Programmatic Use**

**Block Parameter:** InitialConditionForFilter  
**Type:** scalar, vector  
**Default:** 0

**Differentiator — Initial condition for unfiltered derivative**  
0 (default) | scalar | vector

When you use an unfiltered derivative, Simulink uses this parameter to initialize the differentiator at the start of a simulation or at a specified trigger event (see **External reset**). The integrator initial condition and the derivative initial condition determine the initial output of the PID controller block.

The derivative initial condition cannot be NaN or Inf.

**Dependencies**

To use this parameter, set **Time domain** to Discrete-time, clear the **Use filtered derivative** check box, and in the **Initialization** tab, set **Source** to internal.

**Programmatic Use**

**Block Parameter:** DifferentiatorICPrevScaledInput  
**Type:** scalar, vector
**Default:** 0

**Initial condition setting — Location at which initial condition is applied**

*State (most efficient) (default)* | *Output*

Use this parameter to specify whether to apply the **Integrator Initial condition** and **Filter Initial condition** parameter to the corresponding block state or output. You can change this parameter at the command line only, using **set_param** to set the **InitialConditionSetting** parameter of the block.

**State (most efficient)**

- Use this option in all situations except when the block is in a triggered subsystem or a function-call subsystem and simplified initialization mode is enabled.

**Output**

- Use this option when the block is in a triggered subsystem or a function-call subsystem and simplified initialization mode is enabled.

For more information about the **Initial condition setting** parameter, see the Discrete-Time Integrator block.

This parameter is only accessible through programmatic use.

**Programmatic Use**

**Block Parameter:** InitialConditionSetting

**Type:** string, character vector

**Values:** "state", "output"

**Default:** "state"

**External reset — Trigger for resetting integrator and filter values**

*none (default) | rising | falling | either | level*

Specify the trigger condition that causes the block to reset the integrator and filter to initial conditions. (If **Use filtered derivative** is not selected, the trigger resets the integrator and differentiator to initial conditions.) Selecting any option other than none enables the **Reset** port on the block for the external reset signal.

**none**

- The integrator and filter (or differentiator) outputs are set to initial conditions at the beginning of simulation, and are not reset during simulation.

**rising**

- Reset the outputs when the reset signal has a rising edge.
falling
Reset the outputs when the reset signal has a falling edge.
either
Reset the outputs when the reset signal either rises or falls.
level
Reset the outputs when the reset signal either:
• Is nonzero at the current time step
• Changes from nonzero at the previous time step to zero at the current time step

This option holds the outputs to the initial conditions while the reset signal is nonzero.

Dependencies
To enable this parameter, set Controller to a type that has derivative or integral action.

Programmatic Use
Block Parameter: ExternalReset
Type: string, character vector
Values: "none", "rising", "falling", "either", "level"
Default: "none"

Ignore reset when linearizing — Force linearization to ignore reset
off (default) | on

Select to force Simulink and Simulink Control Design linearization commands to ignore any reset mechanism specified in the External reset parameter. Ignoring reset states allows you to linearize a model around an operating point even if that operating point causes the block to reset.

Programmatic Use
Block Parameter: IgnoreLimit
Type: string, character vector
Values: "off", "on"
Default: "off"

Enable tracking mode — Activate signal tracking
off (default) | on

Signal tracking lets the block output follow a tracking signal that you provide at the TR port. When signal tracking is active, the difference between the tracking signal and the
block output is fed back to the integrator input with a gain $K_t$, specified by the **Tracking gain ($K_t$)** parameter. Signal tracking has several applications, including bumpless control transfer and avoiding windup in multiloop control structures.

**Bumpless control transfer**

Use signal tracking to achieve bumpless control transfer in systems that switch between two controllers. Suppose you want to transfer control between a PID controller and another controller. To do so, connecting the controller output to the **TR** input as shown in the following illustration.

For more information, see “Bumpless Control Transfer with a Two-Degree-of-Freedom PID Controller” on page 14-129.

**Multiloop control**

Use signal tracking to prevent block windup in multiloop control approaches. For an example illustrating this approach with a 1DOF PID controller, see “Prevent Block Windup in Multiloop Control” on page 14-127.

**Dependencies**

To enable this parameter, set **Controller** to a type that has integral action.
**Programmatic Use**
**Block Parameter:** TrackingMode  
**Type:** string, character vector  
**Values:** "off", "on"  
**Default:** "off"

**Tracking coefficient (Kt) — Gain of signal-tracking feedback loop**  
1 (default) | scalar

When you select **Enable tracking mode**, the difference between the signal TR and the block output is fed back to the integrator input with a gain Kt. Use this parameter to specify the gain in that feedback loop.

**Dependencies**

To enable this parameter, select **Enable tracking mode**.

**Programmatic Use**
**Block Parameter:** Kt  
**Type:** scalar  
**Default:** 1

**Output saturation**

**Limit Output — Limit block output to specified saturation values**  
off (default) | on

Activating this option limits the block output internally to the block, so that you do not need a separate Saturation block after the controller. It also allows you to activate the anti-windup mechanism built into the block (see the **Anti-windup method** parameter). Specify the saturation limits using the **Lower saturation limit** and **Upper saturation limit** parameters.

**Programmatic Use**
**Block Parameter:** LimitOutput  
**Type:** string, character vector  
**Values:** "off", "on"  
**Default:** "off"

**Upper limit — Upper saturation limit for block output**  
Inf (default) | scalar
Specify the upper limit for the block output. The block output is held at the **Upper saturation limit** whenever the weighted sum of the proportional, integral, and derivative actions exceeds that value.

**Dependencies**

To enable this parameter, select **Limit output**.

**Programmatic Use**

**Block Parameter:** UpperSaturationLimit  
**Type:** scalar  
**Default:** Inf

**Lower limit — Lower saturation limit for block output**

-Inf (default) | scalar

Specify the lower limit for the block output. The block output is held at the **Lower saturation limit** whenever the weighted sum of the proportional, integral, and derivative actions goes below that value.

**Dependencies**

To enable this parameter, select **Limit output**.

**Programmatic Use**

**Block Parameter:** LowerSaturationLimit  
**Type:** scalar  
**Default:** -Inf

**Ignore saturation when linearizing — Force linearization to ignore output limits**

off (default) | on

Force Simulink and Simulink Control Design linearization commands to ignore block output limits specified in the **Upper limit** and **Lower limit** parameters. Ignoring output limits allows you to linearize a model around an operating point even if that operating point causes the block to exceed the output limits.

**Dependencies**

To enable this parameter, select the Limit output parameter.

**Programmatic Use**

**Block Parameter:** LinearizeAsGain
Type: string, character vector
Values: "off", "on"
Default: "off"

Anti-windup method — Integrator anti-windup method
none (default) | back-calculation | clamping

When you select Limit output and the weighted sum of the controller components exceeds the specified output limits, the block output holds at the specified limit. However, the integrator output can continue to grow (integrator windup), increasing the difference between the block output and the sum of the block components. In other words, the internal signals in the block can be unbounded even if the output appears bounded by saturation limits. Without a mechanism to prevent integrator windup, two results are possible:

- If the sign of the signal entering the integrator never changes, the integrator continues to integrate until it overflows. The overflow value is the maximum or minimum value for the data type of the integrator output.

- If the sign of the signal entering the integrator changes once the weighted sum has grown beyond the output limits, it can take a long time to unwind the integrator and return the weighted sum within the block saturation limit.

In either case, controller performance can suffer. To combat the effects of windup without an anti-windup mechanism, it may be necessary to detune the controller (for example, by reducing the controller gains), resulting in a sluggish controller. To avoid this problem, activate an anti-windup mechanism using this parameter.

none
Do not use an anti-windup mechanism.

back-calculation

Unwind the integrator when the block output saturates by feeding back to the integrator the difference between the saturated and unsaturated control signal. The following diagram represents the back-calculation feedback circuit for a continuous-time controller. To see the actual feedback circuit for your controller configuration, right-click on the block and select Mask > Look Under Mask.
Use the **Back-calculation coefficient (Kb)** parameter to specify the gain of the anti-windup feedback circuit. It is usually satisfactory to set $K_b = I$, or for controllers with derivative action, $K_b = \sqrt{I*D}$. Back-calculation can be effective for plants with relatively large dead time [1].

**Clamping**

Integration stops when the sum of the block components exceeds the output limits and the integrator output and block input have the same sign. Integration resumes when the sum of the block components exceeds the output limits and the integrator output and block input have opposite sign. Clamping is sometimes referred to as conditional integration.

Clamping can be useful for plants with relatively small dead times, but can yield a poor transient response for large dead times [1].

**Dependencies**

To enable this parameter, select the **Limit output** parameter.

**Programmatic Use**

**Block Parameter:** AntiWindupMode  
**Type:** string, character vector  
**Values:** "none", "back-calculation", "clamping"
Default: "none"

**Back-calculation coefficient** *(Kb)* — Gain coefficient of anti-windup feedback loop
1 (default) | scalar

The back-calculation anti-windup method unwinds the integrator when the block output saturates. It does so by feeding back to the integrator the difference between the saturated and unsaturated control signal. Use the **Back-calculation coefficient** *(Kb)* parameter to specify the gain of the anti-windup feedback circuit. For more information, see the **Anti-windup method** parameter.

**Dependencies**

To enable this parameter, select the **Limit output** parameter, and set the **Anti-windup method** parameter to **back-calculation**.

**Programmatic Use**

**Block Parameter:** Kb  
**Type:** scalar  
**Default:** 1

**Data Types**

The parameters in this tab are primarily of use in fixed-point code generation using Fixed-Point Designer. They define how numeric quantities associated with the block are stored and processed when you generate code.

If you need to configure data types for fixed-point code generation, click **Open Fixed-Point Tool** and use that tool to configure the rest of the parameters in the tab. For information about using Fixed-Point Tool, see “Autoscaling Data Objects Using the Fixed-Point Tool” (Fixed-Point Designer).

After you use Fixed-Point Tool, you can use the parameters in this tab to make adjustments to fixed-point data-type settings if necessary. For each quantity associated with the block, you can specify:

- Floating-point or fixed-point data type, including whether the data type is inherited from upstream values in the block.
- The minimum and maximum values for the quantity, which determine how the quantity is scaled for fixed-point representation.
For assistance in selecting appropriate values, click to open the Data Type Assistant for the corresponding quantity. For more information, see “Specify Data Types Using Data Type Assistant”.

The specific quantities listed in the Data Types tab vary depending on how you configure the PID controller block. In general, you can configure data types for the following types of quantities:

- **Product output** — Stores the result of a multiplication carried out under the block mask. For example, **P product output** stores the output of the gain block that multiplies the block input with the proportional gain \( P \).
- **Parameter** — Stores the value of a numeric block parameter, such as \( P \), \( I \), or \( D \).
- **Block output** — Stores the output of a block that resides under the PID controller block mask. For example, use **Integrator output** to specify the data type of the
output of the block called Integrator. This block resides under the mask in the Integrator subsystem, and computes integrator term of the controller action.

- Accumulator — Stores values associated with a sum block. For example, **SumI2 Accumulator** sets the data type of the accumulator associated with the sum block SumI2. This block resides under the mask in the Back Calculation subsystem of the Anti-Windup subsystem.

In general, you can find the block associated with any listed parameter by looking under the PID Controller block mask and examining its subsystems. You can also use the Model Explorer to search under the mask for the listed parameter name, such as SumI2. (See “Search and Edit Using Model Explorer”.)

**Matching Input and Internal Data Types**

By default, all data types in the block are set to **Inherit: Inherit via internal rule**. With this setting, Simulink chooses data types to balance numerical accuracy, performance, and generated code size, while accounting for the properties of the embedded target hardware.

Under some conditions, incompatibility can occur between data types within the block. For instance, in continuous time, the Integrator block under the mask can accept only signals of type double. If the block input signal is a type that cannot be converted to double, such as uint16, the internal rules for type inheritance generate an error when you generate code.

To avoid such errors, you can use the Data Types settings to force a data type conversion. For instance, you can explicitly set **P product output**, **I product output**, and **D product output** to double, ensuring that the signals reaching the continuous-time integrators are of type double.

In general, it is not recommended to use the block in continuous time for code generation applications. However, similar data type errors can occur in discrete time, if you explicitly set some values to data types that are incompatible with downstream signal constraints within the block. In such cases, use the Data Types settings to ensure that all data types are internally compatible.

**Fixed-Point Operational Parameters**

**Integer rounding mode — Rounding mode for fixed-point operations**

- Floor (default)
- Ceiling
- Convergent
- Nearest
- Round
- Simplest
- Zero
Specify the rounding mode for fixed-point operations. For more information, see “Rounding” (Fixed-Point Designer).

Block parameters always round to the nearest representable value. To control the rounding of a block parameter, enter an expression using a MATLAB rounding function into the mask field.

**Programmatic Use**

**Block Parameter:** RndMeth  
**Type:** character vector  
**Values:** 'Ceiling' | 'Convergent' | 'Floor' | 'Nearest' | 'Round' | 'Simplest' | 'Zero'  
**Default:** 'Floor'

**Saturate on integer overflow — Method of overflow action**

off (default) | on  

Specify whether overflows saturate or wrap.

- **off** — Overflows wrap to the appropriate value that the data type can represent.
  
  For example, the number 130 does not fit in a signed 8-bit integer and wraps to -126.

- **on** — Overflows saturate to either the minimum or maximum value that the data type can represent.
  
  For example, an overflow associated with a signed 8-bit integer can saturate to -128 or 127.

**Tip**

- Consider selecting this check box when your model has a possible overflow and you want explicit saturation protection in the generated code.

- Consider clearing this check box when you want to optimize efficiency of your generated code.

  Clearing this check box also helps you to avoid overspecifying how a block handles out-of-range signals. For more information, see “Check for Signal Range Errors”.

- When you select this check box, saturation applies to every internal operation on the block, not just the output or result.

- In general, the code generation process can detect when overflow is not possible. In this case, the code generator does not produce saturation code.
Programmatic Use

Block Parameter: SaturateOnIntegerOverflow
Type: character vector
Values: 'off' | 'on'
Default: 'off'

Lock data type settings against changes by the fixed-point tools — Prevent fixed-point tools from overriding data types
off (default) | on

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on this block. For more information, see “Lock the Output Data Type Setting” (Fixed-Point Designer).

Programmatic Use

Block Parameter: LockScale
Type: character vector
Values: 'off' | 'on'
Default: 'off'

State Attributes

The parameters in this tab are primarily of use in code generation.

State name (e.g., 'position') — Name for continuous-time filter and integrator states
'' (default) | character vector

Assign a unique name to the state associated with the integrator or the filter, for continuous-time PID controllers. (For information about state names in a discrete-time PID controller, see the State name parameter.) The state name is used, for example:

- For the corresponding variable in generated code
- As part of the storage name when logging states during simulation
- For the corresponding state in a linear model obtain by linearizing the block

A valid state name begins with an alphabetic or underscore character, followed by alphanumeric or underscore characters.

Dependencies

To enable this parameter, set Time domain to Continuous-time.
Programmatic Use
Parameter: IntegratorContinuousStateAttributes, FilterContinuousStateAttributes
Type: character vector
Default: ''

State name — Names for discrete-time filter and integrator states
empty string (default) | string | character vector

Assign a unique name to the state associated with the integrator or the filter, for discrete-time PID controllers. (For information about state names in a continuous-time PID controller, see the State name (e.g., 'position') parameter.)

A valid state name begins with an alphabetic or underscore character, followed by alphanumeric or underscore characters. The state name is used, for example:

- For the corresponding variable in generated code
- As part of the storage name when logging states during simulation
- For the corresponding state in a linear model obtain by linearizing the block

For more information about the use of state names in code generation, see “Apply Storage Classes to Individual Signal, State, and Parameter Data Elements” (Simulink Coder).

Dependencies
To enable this parameter, set Time domain to Discrete-time.

Programmatic Use
Parameter: IntegratorStateIdentifier, FilterStateIdentifier
Type: string, character vector
Default: ""

State name must resolve to Simulink signal object — Require that state name resolve to a signal object
off (default) | on

Select this parameter to require that the discrete-time integrator or filter state name resolves to a Simulink signal object.

Dependencies
To enable this parameter for the discrete-time integrator or filter state:
1. Set **Time domain** to **Discrete-time**.
2. Specify a value for the integrator or filter **State name**.
3. Set the model configuration parameter **Signal resolution** to a value other than None.

Selecting this check box disables **Code generation storage class** for the corresponding integrator or filter state.

**Programmatic Use**

**Block Parameter:** IntegratorStateMustResolveToSignalObject, FilterStateMustResolveToSignalObject

**Type:** string, character vector

**Values:** "off", "on"

**Default:** "off"

---

**Code generation storage class — Storage class for code generation**

*Auto (default) | ExportedGlobal | ImportedExtern | ImportedExternPointer*

Select state storage class for code generation. If you do not need to interface to external code, select **Auto**.

For more information, see “Apply Storage Classes to Individual Signal, State, and Parameter Data Elements” (Simulink Coder) and “Apply Built-In and Customized Storage Classes to Data Elements” (Embedded Coder).

**Dependencies**

To enable this parameter for the discrete-time integrator or filter state:

1. Set **Time domain** to **Discrete-time**.
2. Specify a value for the integrator or filter **State name**.
3. Set the model configuration parameter **Signal resolution** to a value other than None.

**Programmatic Use**

**Block Parameter:** IntegratorRTWStateStorageClass, FilterRTWStateStorageClass

**Type:** string, character vector

**Values:** "Auto", "ExportedGlobal", "ImportedExtern" | "ImportedExternPointer"

**Default:** "Auto"
**Code generation storage type qualifier — Storage type qualifier**
empty string (default) | character vector | "const" | "volatile" | ...

Specify a storage type qualifier such as const or volatile.

**Note** This parameter will be removed in a future release. To apply storage type qualifiers to data, use custom storage classes and memory sections. Unless you use an ERT-based code generation target with Embedded Coder, custom storage classes and memory sections do not affect the generated code.

**Dependencies**
To enable this parameter, set **Code generation storage class** to any value other than Auto.

**Programmatic Use**
**Block Parameter:**
- IntegratorRTWStateStorageTypeQualifier
- FilterRTWStateStorageTypeQualifier
**Type:** string, character vector
**Values:** "", "const", "volatile"
**Default:** ""

**Block Characteristics**

<table>
<thead>
<tr>
<th>Data Types</th>
<th>double</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
More About

**Decomposition of 2-DOF PID Controllers**

A 2-DOF PID controller can be interpreted as a PID controller with a prefilter, or a PID controller with a feedforward element.

**Prefilter Decomposition**

In parallel form, a two-degree-of-freedom PID controller can be equivalently modeled by the following block diagram, where $C$ is a single degree-of-freedom PID controller and $F$ is a prefilter on the reference signal.

![Block Diagram]

$Ref$ is the reference signal, $y$ is the feedback from the measured system output, and $u$ is the controller output. For a continuous-time 2-DOF PID controller in parallel form, the transfer functions for $F$ and $C$ are

\[
F_{par}(s) = \frac{(bP + cDN)s^2 + (bPN + I)s + IN}{(P + DN)s^2 + (PN + I)s + IN},
\]

\[
C_{par}(s) = \frac{(P + DN)s^2 + (PN + I)s + IN}{s(s + N)},
\]

where $b$ and $c$ are the setpoint weights.

For a 2-DOF PID controller in ideal form, the transfer functions are

\[
F_{id}(s) = \frac{(b + cDN)s^2 + (bN + I)s + IN}{(1 + DN)s^2 + (N + I)s + IN},
\]

\[
C_{id}(s) = P\frac{(1 + DN)s^2 + (N + I)s + IN}{s(s + N)}.
\]
A similar decomposition applies for a discrete-time 2-DOF controller.

**Feedforward Decomposition**

Alternatively, the parallel two-degree-of-freedom PID controller can be modeled by the following block diagram.

![Block Diagram](image)

In this realization, $Q$ acts as feed-forward conditioning on the reference signal. For a continuous-time 2-DOF PID controller in parallel form, the transfer function for $Q$ is

$$Q_{par}(s) = \frac{[(b - 1)P + (c - 1)DN)s + (b - 1)PN}{s + N}.$$  

For a 2-DOF PID controller in ideal form, the transfer function is

$$Q_{id}(s) = P(\frac{(b - 1) + (c - 1)DN)s + (b - 1)N}{s + N}.$$  

The transfer functions for $C$ are the same as in the filter decomposition.

A similar decomposition applies for a discrete-time 2-DOF controller.

**References**

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

For continuous-time PID controllers (Time domain set to Continuous-time):

• Consider using “Model Discretizer” to map continuous-time blocks to discrete equivalents that support code generation. To access Model Discretizer, in the Apps tab, under Control Systems, click Model Discretizer.
• Not recommended for production code.

For discrete-time PID controllers (Time domain set to Discrete-time):

• Depends on absolute time when placed inside a triggered subsystem hierarchy.
• Generated code relies on memcpy or memset functions (string.h) under certain conditions.

PLC Code Generation
Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

Fixed-point code generation is supported for discrete-time PID controllers only (Time domain set to Discrete-time).

See Also
Discrete Derivative | Discrete PID Controller | Discrete-Time Integrator | Gain | PID Controller (2DOF)

Introduced in R2009b
Discrete State-Space

Implement discrete state-space system
Library: Simulink / Discrete

Description

Block Behavior for Non-Empty Matrices

The Discrete State-Space block implements the system described by

\[ x(n+1) = Ax(n) + Bu(n) \]
\[ y(n) = Cx(n) + Du(n), \]

where \( u \) is the input, \( x \) is the state, and \( y \) is the output. The matrix coefficients must have these characteristics, as illustrated in the following diagram:

- \( A \) must be an \( n \)-by-\( n \) matrix, where \( n \) is the number of states.
- \( B \) must be an \( n \)-by-\( m \) matrix, where \( m \) is the number of inputs.
- \( C \) must be an \( r \)-by-\( n \) matrix, where \( r \) is the number of outputs.
- \( D \) must be an \( r \)-by-\( m \) matrix.

The block accepts one input and generates one output. The width of the input vector is the number of columns in the \( B \) and \( D \) matrices. The width of the output vector is the number of rows in the \( C \) and \( D \) matrices. To define the initial state vector, use the Initial conditions parameter.

To specify a vector or matrix of zeros for \( A, B, C, D \), or Initial conditions, use the zeros function.
Block Behavior for Empty Matrices

When the matrices $A$, $B$, and $C$ are empty (for example, $[]$), the functionality of the block becomes $y(n) = Du(n)$. If the **Initial conditions** vector is also empty, the block uses an initial state vector of zeros.

Ports

Input

**Port 1 — Input signal**

*scalar | vector*

Input vector, where the width equals the number of columns in the $B$ and $D$ matrices. For more information, see “Description” on page 1-599.

**Tip** For integer and fixed-point input signals, use the Fixed-Point State-Space block.

Data Types: *single | double*

Output

**Port 1 — Output vector**

*scalar | vector*

Output vector, with width equal to the number of rows in the $C$ and $D$ matrices. For more information, see “Description” on page 1-599.

Data Types: *single | double*

Parameters

Main

**A — Matrix coefficient A**

*1 (default) | scalar | vector | matrix*
Specify the matrix coefficient $A$, as a real-valued $n$-by-$n$ matrix, where $n$ is the number of states. For more information on the matrix coefficients, see “Description” on page 1-599.

**Programmatic Use**

*Block Parameter:* $A$

**Type:** character vector

**Values:** scalar | vector | matrix

**Default:** '1'

---

### B — Matrix coefficient $B$

1 (default) | scalar | vector | matrix

Specify the matrix coefficient $B$, as a real-valued $n$-by-$m$ matrix, where $n$ is the number of states, and $m$ is the number of inputs. For more information on the matrix coefficients, see “Description” on page 1-599.

**Programmatic Use**

*Block Parameter:* $B$

**Type:** character vector

**Values:** scalar | vector | matrix

**Default:** '1'

---

### C — Matrix coefficient, $C$

1 (default) | scalar | vector | matrix

Specify the matrix coefficient $C$, as a real-valued $r$-by-$n$ matrix, where $r$ is the number of outputs, and $n$ is the number of states. For more information on the matrix coefficients, see “Description” on page 1-599.

**Programmatic Use**

*Block Parameter:* $C$

**Type:** character vector

**Values:** scalar | vector | matrix

**Default:** '1'

---

### D — Matrix coefficient, $D$

1 (default) | scalar | vector | matrix

Specify the matrix coefficient $D$, as a real-valued $r$-by-$m$ matrix, where $r$ is the number of outputs, and $m$ is the number of inputs. For more information on the matrix coefficients, see “Description” on page 1-599.
**Programmatic Use**

**Block Parameter:** `D`

Type: character vector  
Values: scalar | vector | matrix  
Default: `'1'`

**Initial conditions — Initial state vector**

0 (default) | scalar | vector

Specify the initial state vector as a scalar or vector. The initial state vector cannot include `inf` or `NaN` values.

**Programmatic Use**

**Block Parameter:** `InitialCondition`

Type: character vector  
Values: scalar | vector  
Default: `'0'`

**Sample time (–1 for inherited) — Interval between samples**

-1 (default) | scalar | vector

Specify the time interval between samples. See “Specify Sample Time”.

**Programmatic Use**

**Block Parameter:** `SampleTime`

Type: character vector  
Values: scalar | vector  
Default: `' -1'`

**State Attributes**

**State name — Unique name for block state**

' ' (default) | alphanumeric string

Use this parameter to assign a unique name to the block state. The default is ' '. When this field is blank, no name is assigned. When using this parameter, remember these considerations:

- A valid identifier starts with an alphabetic or underscore character, followed by alphanumeric or underscore characters.
- The state name applies only to the selected block.
This parameter enables **State name must resolve to Simulink signal object** when you click **Apply**.

For more information, see “Apply Storage Classes to Individual Signal, State, and Parameter Data Elements” (Simulink Coder).

**Programmatic Use**

**Block Parameter:** StateName  
**Type:** character vector  
**Values:** unique name  
**Default:** ''

**State name must resolve to Simulink signal object — Require state name resolve to a signal object**

*off* (default) | *on*

Select this check box to require that the state name resolves to a Simulink signal object.

**Dependencies**

To enable this parameter, specify a value for **State name**. This parameter appears only if you set the model configuration parameter **Signal resolution** to a value other than **None**.

Selecting this check box disables **Code generation storage class**.

**Programmatic Use**

**Block Parameter:** StateMustResolveToSignalObject  
**Type:** character vector  
**Values:** 'off' | 'on'  
**Default:** 'off'

**Signal object class — Custom storage class package name**

Simulink.Signal (default) | <StorageClass.PackageName>

Choose a custom storage class package by selecting a signal object class that the target package defines. For example, to apply custom storage classes from the built-in package mpt, select mpt.Signal. Unless you use an ERT-based code generation target with Embedded Coder, custom storage classes do not affect the generated code.

If the class that you want does not appear in the list, select **Customize class lists**. For instructions, see “Target Class Does Not Appear in List of Signal Object Classes” (Embedded Coder).
For information about storage classes, see “Apply Storage Classes to Individual Signal, State, and Parameter Data Elements” (Simulink Coder). For information about custom storage classes, see “Apply Built-In and Customized Storage Classes to Data Elements” (Embedded Coder).

**Programmatic Use**

**Block Parameter:** StateSignalObject  
**Type:** character vector  
**Values:** 'Simulink.Signal' | '<StorageClass.PackageName>'  
**Default:** 'Simulink.Signal'

**Code generation storage class — State storage class for code generation**  
Auto (default) | Model default | ExportedGlobal | ImportedExtern | ImportedExternPointer | BitField (Custom) | Model default | ExportToFile (Custom) | ImportFromFile (Custom) | FileScope (Custom) | AutoScope (Custom) | Struct (Custom) | GetSet (Custom) | Reusable (Custom)

Select state storage class for code generation.

- **Auto** is the appropriate storage class for states that you do not need to interface to external code.
- **StorageClass** applies the storage class or custom storage class that you select from the list. For information about storage classes, see “Apply Storage Classes to Individual Signal, State, and Parameter Data Elements” (Simulink Coder). For information about custom storage classes, see “Apply Built-In and Customized Storage Classes to Data Elements” (Embedded Coder).

Use **Signal object class** to select custom storage classes from a package other than Simulink.

**Dependencies**

To enable this parameter, specify a value for **State name**.

**Programmatic Use**

**Block Parameter:** StateStorageClass  
**Type:** character vector  
**Values:** 'Auto' | 'Model default' | 'ExportedGlobal' | 'ImportedExtern' | 'ImportedExternPointer' | 'Custom' | ...  
**Default:** 'Auto'

**TypeQualifier — Storage type qualifier**  
'' (default) | const | volatile | ...
Specify a storage type qualifier such as const or volatile.

**Note** TypeQualifier will be removed in a future release. To apply storage type qualifiers to data, use custom storage classes and memory sections. Unless you use an ERT-based code generation target with Embedded Coder, custom storage classes and memory sections do not affect the generated code.

During simulation, the block uses the following values:

- The initial value of the signal object to which the state name is resolved
- Minimum and Maximum values of the signal object

For more information, see “Data Objects”.

**Dependencies**

To enable this parameter, set **Code generation storage class** to ExportedGlobal, ImportedExtern, ImportedExternPointer, or Model default. This parameter is hidden unless you previously set its value.

**Programmatic Use**

**Block Parameter:** RTWStateStorageTypeQualifier  
**Type:** character vector  
**Values:** '' | 'const' | 'volatile' | ...  
**Default:** ''

### Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>double</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
<td></td>
</tr>
</tbody>
</table>
Zero-Crossing Detection | no

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

Generated code relies on memcpy or memset functions (string.h) under certain conditions.

PLC Code Generation
Generate Structured Text code using Simulink® PLC Coder™.

See Also
Fixed-Point State-Space | State-Space

Topics
“Apply Storage Classes to Individual Signal, State, and Parameter Data Elements” (Simulink Coder)
“Apply Built-In and Customized Storage Classes to Data Elements” (Embedded Coder)
“Data Objects”

Introduced before R2006a
Discrete-Time Integrator

Perform discrete-time integration or accumulation of signal

Library: Simulink / Commonly Used Blocks
        Simulink / Discrete
        HDL Coder / Discrete
        HDL Coder / HDL Floating Point Operations

Description

Use the Discrete-Time Integrator block in place of the Integrator block to create a purely discrete model. With the Discrete-Time Integrator block, you can:

- Define initial conditions on the block dialog box or as input to the block
- Define an input gain (K) value
- Output the block state
- Define upper and lower limits on the integral
- Reset the state with an additional reset input

Output Equations

With the first time step, block state \( n = 0 \), with either initial output \( y(0) = IC \) or initial state \( x(0) = IC \), depending on the Initial condition setting parameter value.

For a given step \( n > 0 \) with simulation time \( t(n) \), Simulink updates output \( y(n) \) as follows:

- Forward Euler method:
  \[
  y(n) = y(n-1) + K*[t(n) - t(n-1)]*u(n-1)
  \]
- Backward Euler method:
  \[
  y(n) = y(n-1) + K*[t(n) - t(n-1)]*u(n)
  \]
- Trapezoidal method:
\[ y(n) = y(n-1) + K * [t(n) - t(n-1)] * [u(n) + u(n-1)] / 2 \]

Simulink automatically selects a state-space realization of these output equations depending on the block sample time, which can be explicit or triggered. When using explicit sample time, \( t(n) - t(n-1) \) reduces to the sample time \( T \) for all \( n > 0 \).

**Integration and Accumulation Methods**

This block can integrate or accumulate a signal using a forward Euler, backward Euler, or trapezoidal method. Assume that \( u \) is the input, \( y \) is the output, and \( x \) is the state. For a given step \( n \), Simulink updates \( y(n) \) and \( x(n+1) \). In integration mode, \( T \) is the block sample time (delta \( T \) in the case of triggered sample time). In accumulation mode, \( T = 1 \). The block sample time determines when the output is computed but not the output value. \( K \) is the gain value. Values clip according to upper or lower limits.

**Forward Euler Method**

Forward Euler method (default), also known as forward rectangular, or left-hand approximation

The software approximates \( 1/s \) as \( T / (z - 1) \). The expressions for the output of the block at step \( n \) are:

\[
\begin{align*}
x(n+1) &= x(n) + K * T * u(n) \\
y(n) &= x(n)
\end{align*}
\]

The block uses these steps to compute the output:

**Step 0:**

\[
\begin{align*}
y(0) &= IC \text{ (clip if necessary)} \\
x(1) &= y(0) + K * T * u(0)
\end{align*}
\]

**Step 1:**

\[
\begin{align*}
y(1) &= x(1) \\
x(2) &= x(1) + K * T * u(1)
\end{align*}
\]

**Step n:**

\[
\begin{align*}
y(n) &= x(n) \\
x(n+1) &= x(n) + K * T * u(n) \text{ (clip if necessary)}
\end{align*}
\]

Using this method, input port 1 does not have direct feedthrough.

**Backward Euler Method**

Backward Euler method, also known as backward rectangular or right-hand approximation
The software approximates $1/s$ as $Tz/(z-1)$. The resulting expression for the output of the block at step $n$ is

$$y(n) = y(n-1) + K*T*u(n).$$

Let $x(n) = y((n)-1)$. The block uses these steps to compute the output.

- If the parameter **Initial condition setting** is set to **Output** or **Auto** for triggered and function-call subsystems:

  **Step 0:**
  - $y(0) = IC$ (clipped if necessary)
  - $x(1) = y(0)$

- If the parameter **Initial condition setting** is set to **Auto** for non-triggered subsystems:

  **Step 0:**
  - $x(0) = IC$ (clipped if necessary)
  - $x(1) = y(0) = x(0) + K*T*u(0)$

  **Step 1:**
  - $y(1) = x(1) + K*T*u(1)$
  - $x(2) = y(1)$

  **Step $n$:**
  - $y(n) = x(n) + K*T*u(n)$
  - $x(n+1) = y(n)$

Using this method, input port 1 has direct feedthrough.

**Trapezoidal Method**

For this method, the software approximates $1/s$ as $T/2*(z+1)/(z-1)$.

When $T$ is fixed (equal to the sampling period), the expressions to compute the output are:

$$x(n) = y(n-1) + \frac{K*T}{2}u(n-1)$$
$$y(n) = x(n) + \frac{K*T}{2}u(n)$$

- If the parameter **Initial condition setting** is set to **Output** or **Auto** for triggered and function-call subsystems:

  **Step 0:**
  - $y(0) = IC$ (clipped if necessary)
  - $x(1) = y(0) + K*T/2*u(0)$

- If the parameter **Initial condition setting** is set to **Auto** for non-triggered subsystems:
Step 0: 
\[ x(0) = \text{IC (clipped if necessary)} \]
\[ y(0) = x(0) + K\times T/2\times u(0) \]
\[ x(1) = y(0) + K\times T/2\times u(0) \]

Step 1: 
\[ y(1) = x(1) + K\times T/2\times u(1) \]
\[ x(2) = y(1) + K\times T/2\times u(1) \]

Step n: 
\[ y(n) = x(n) + K\times T/2\times u(n) \]
\[ x(n+1) = y(n) + K\times T/2\times u(n) \]

Here, \( x(n+1) \) is the best estimate of the next output. It is not the same as the state, in that \( x(n) \) is not equal to \( y(n) \).

Using this method, input port 1 has direct feedthrough.

**When T is a Variable**

When \( T \) is a variable (for example, obtained from the triggering times), the block uses these steps to compute the output.

- If the parameter **Initial condition setting** is set to **Output** or **Auto** for triggered and function-call subsystems:

  Step 0: 
  \[ y(0) = \text{IC (clipped if necessary)} \]
  \[ x(1) = y(0) \]

- If the parameter **Initial condition setting** is set to **Auto** for non-triggered subsystems:

  Step 0: 
  \[ x(0) = \text{IC (clipped if necessary)} \]
  \[ x(1) = y(0) = x(0) + K\times T/2\times u(0) \]

  Step 1: 
  \[ y(1) = x(1) + T/2\times(u(1) + u(0)) \]
  \[ x(2) = y(1) \]

  Step n: 
  \[ y(n) = x(n) + T/2\times(u(n) + u(n-1)) \]
  \[ x(n+1) = y(n) \]

**Define Initial Conditions**

You can define the initial conditions as a parameter on the block dialog box or input them from an external signal:

- To define the initial conditions as a block parameter, set the **Initial condition source** parameter to **internal** and enter the value in the **Initial condition** text box.
• To provide the initial conditions from an external source, set the Initial condition source parameter to external. An additional input port appears on the block.

When to Use the State Port

Use the state port instead of the output port:

• When the output of the block is fed back into the block through the reset port or the initial condition port, causing an algebraic loop. For an example, see the sldemo_bounce_two_integrators model.
• When you want to pass the state from one conditionally executed subsystem to another, which can cause timing problems. For an example, see the sldemo_clutch model.

You can work around these problems by passing the state through the state port rather than the output port. Simulink generates the state at a slightly different time from the output, which protects your model from these problems. To output the block state, select the Show state port check box. The state port appears on the top of the block.
Limit the Integral

To keep the output within certain levels, select the Limit output check box and enter the limits in the corresponding text box. Doing so causes the block to function as a limited integrator. When the output reaches the limits, the integral action turns off to prevent integral windup. During a simulation, you can change the limits but you cannot change whether the output is limited. The table shows how the block determines output.

<table>
<thead>
<tr>
<th>Integral</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Less than or equal to the <strong>Lower saturation limit</strong> and the input is negative</td>
<td>Held at the <strong>Lower saturation limit</strong></td>
</tr>
<tr>
<td>Between the <strong>Lower saturation limit</strong> and the <strong>Upper saturation limit</strong></td>
<td>The integral</td>
</tr>
<tr>
<td>Greater than or equal to the <strong>Upper saturation limit</strong> and the input is positive</td>
<td>Held at the <strong>Upper saturation limit</strong></td>
</tr>
</tbody>
</table>

To generate a signal that indicates when the state is being limited, select the Show saturation port check box. A new saturation port appears below the block output port.

The saturation signal has one of three values:

- 1 indicates that the upper limit is being applied.
- 0 indicates that the integral is not limited.
- -1 indicates that the lower limit is being applied.

Reset the State

The block resets its state to the specified initial condition, based on an external signal. To cause the block to reset its state, select one of the **External reset** parameter options. A reset port appears that indicates the reset trigger type.
The reset port has direct feedthrough. If the block output feeds back into this port, either directly or through a series of blocks with direct feedthrough, an algebraic loop results. To resolve this loop, feed the output of the block state port into the reset port instead. To access the block state, select the **Show state port** check box.

### Reset Trigger Types

The **External reset** parameter lets you determine the attribute of the reset signal that triggers the reset. The trigger options include:

- **rising** – Resets the state when the reset signal has a rising edge. For example, this figure shows the effect that a rising reset trigger has on backward Euler integration.

- **falling** — Resets the state when the reset signal has a falling edge. For example, this figure shows the effect that a falling reset trigger has on backward Euler integration.
• **either** — Resets the state when the reset signal rises or falls. For example, the following figure shows the effect that an either reset trigger has on backward Euler integration.

• **level** — Resets and holds the output to the initial condition while the reset signal is nonzero. For example, this figure shows the effect that a level reset trigger has on backward Euler integration.
- **sampled level** — Resets the output to the initial condition when the reset signal is nonzero. For example, this figure shows the effect that a sampled level reset trigger has on backward Euler integration.

The sampled level reset option requires fewer computations, making it more efficient than the level reset option.

**Note** For the Discrete-Time Integrator block, all trigger detections are based on signals with positive values. For example, a signal changing from -1 to 0 is not considered a rising edge, but a signal changing from 0 to 1 is.
Behavior in Simplified Initialization Mode

Simplified initialization mode is enabled when you set **Underspecified initialization detection** to **Simplified** in the Configuration Parameters dialog box. If you use simplified initialization mode, the behavior of the Discrete-Time Integrator block differs from classic initialization mode. The new initialization behavior is more robust and provides more consistent behavior in these cases:

- In algebraic loops
- On enable and disable
- When comparing results using triggered sample time against explicit sample time, where the block is triggered at the same rate as the explicit sample time

Simplified initialization mode enables easier conversion from Continuous-Time Integrator blocks to Discrete-Time Integrator blocks, because the initial conditions have the same meaning for both blocks.

For more information on classic and simplified initialization modes, see “Underspecified initialization detection”.

Enable and Disable Behavior with Initial Condition Setting set to Output

When you use simplified initialization mode with **Initial condition setting** set to **Output** for triggered and function-call subsystems, the enable and disable behavior of the block is simplified as follows.

At disable time \( t_d \):
\[
y(t_d) = y(t_d - 1)
\]

At enable time \( t_e \):

- If the parent subsystem control port has **States when enabling** set to **reset**:
  \[
y(t_e) = IC.
\]
- If the parent subsystem control port has **States when enabling** set to **held**:
  \[
y(t_e) = y(t_d).
\]

The following figure shows this condition.
Iterator Subsystems

When using simplified initialization mode, you cannot place the Discrete-Time Integrator block in an iterator subsystem block.

In simplified initialization mode, Iterator subsystems do not maintain elapsed time. Thus, if a Discrete-Time Integrator block, which needs elapsed time, is placed inside an iterator subsystem block, Simulink reports an error.

Behavior in an Enabled Subsystem Inside a Function-Call Subsystem

Suppose you have a function-call subsystem that includes an enabled subsystem, which contains a Discrete-Time Integrator block. The following behavior applies.
<table>
<thead>
<tr>
<th>Integrator Method</th>
<th>Sample Time Type of Function-Call Trigger Port</th>
<th>Value of $\Delta T$ When Function-Call Subsystem Executes for the First Time After Enabled</th>
<th>Reason for Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>Forward Euler</td>
<td>Triggered</td>
<td>$t - t_{\text{start}}$</td>
<td>When the function-call subsystem executes for the first time, the integrator algorithm uses $t_{\text{start}}$ as the previous simulation time.</td>
</tr>
<tr>
<td>Backward Euler and Trapezoidal</td>
<td>Triggered</td>
<td>$t - t_{\text{previous}}$</td>
<td>When the function-call subsystem executes for the first time, the integrator algorithm uses $t_{\text{previous}}$ as the previous simulation time.</td>
</tr>
<tr>
<td>Forward Euler, Backward Euler, and Trapezoidal</td>
<td>Periodic</td>
<td>Sample time of the function-call generator</td>
<td>In periodic mode, the Discrete-Time Integrator block uses sample time of the function-call generator for $\Delta T$.</td>
</tr>
</tbody>
</table>

**Ports**

**Input**

**Port 1 — Input signal**

scalar | vector | matrix

Input signal, specified as a scalar, vector, or matrix

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point
**IC — Initial conditions of the states**
scalar | vector | matrix

Initial conditions of the states, specified as a finite scalar, vector, or matrix.

**Dependencies**
To enable this port, set **Initial condition source** to external.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

**Output**

**Port_1 — Discrete-time integration or accumulation of input**
scalar | vector | matrix

Discrete-time integration or accumulation of the input signal, specified as a scalar, vector, or matrix.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

**Port_2 — Saturation output**
scalar | vector | matrix

Signal indicating when the state is being limited, specified as a scalar, vector, or matrix. The signal has one of three values:

- 1 indicates that the upper limit is being applied.
- 0 indicates that the integral is not limited.
- -1 indicates that the lower limit is being applied.

**Dependencies**
To enable this port, select the **Show saturation port** check box.

Data Types: single | double | int8

**Port_3 — State output**
scalar | vector | matrix

Block states, output as a scalar, vector, or matrix. By default, the block adds this port to the top of the block icon. Use the state port when:
The output of the block is fed back into the block through the reset port or the initial condition port, causing an algebraic loop. For an example, see the `sldemo_bounce_two_integrators` model.

You want to pass the state from one conditionally executed subsystem to another, which can cause timing problems. For an example, see the `sldemo_clutch` model.

For more information, see “When to Use the State Port” on page 1-611.

**Dependencies**

To enable this port, select the **Show state port** check box.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

**Parameters**

**Main**

**Integrator method — Accumulation method**

Integration: Forward Euler (default) | Integration: Backward Euler | Integration: Trapezoidal | Accumulation: Forward Euler | Accumulation: Backward Euler | Accumulation: Trapezoidal

Specify the integration or accumulation method. See “Output Equations” on page 1-607 and “Integration and Accumulation Methods” on page 1-608 for more information.

**Programmatic Use**

**Block Parameter:** `IntegratorMethod`

**Type:** character vector

**Values:**
- `Integration: Forward Euler`
- `Integration: Backward Euler`
- `Integration: Trapezoidal`
- `Accumulation: Forward Euler`
- `Accumulation: Backward Euler`
- `Accumulation: Trapezoidal`

**Default:** `Integration: Forward Euler`

**Gain value — Value to multiply with integrator input**

1.0 (default) | scalar | vector

Specify a scalar, vector, or matrix by which to multiply the integrator input. Each element of the gain must be a positive real number.
• Specifying a value other than 1.0 (the default) is semantically equivalent to connecting a Gain block to the input of the integrator.

• Valid entries include:
  • `double(1.0)`
  • `single(1.0)`
  • `[1.1 2.2 3.3 4.4]`
  • `[1.1 2.2; 3.3 4.4]`

**Tip** Using this parameter to specify the input gain eliminates a multiplication operation in the generated code. However, this parameter must be nontunable to realize this benefit. If you want to tune the input gain, set this parameter to 1.0 and use an external Gain block to specify the input gain.

**Programmatic Use**

**Block Parameter:** `gainval`  
**Type:** character vector  
**Values:** scalar | vector  
**Default:** '1.0'

**External reset — Select when to reset states to initial conditions**

`none` (default) | `rising` | `falling` | `either` | `level` | `sampled level`

Select the type of trigger event that resets the states to their initial conditions:

• `none` — Do not reset the state to initial conditions.
• `rising` — Reset the state when the reset signal has a rising edge.
• `falling` — Reset the state when the reset signal has a falling edge.
• `either` — Reset the state when the reset signal rises or falls.
• `level` — Reset and hold the output to the initial condition while the reset signal is nonzero.
• `sampled level` — Reset the output to the initial condition when the reset signal is nonzero.

For more information, see “Reset the State” on page 1-612 and “Reset Trigger Types” on page 1-613.
**Programmatic Use**  
**Block Parameter:** ExternalReset  
**Type:** character vector  
**Values:** 'none' | 'rising' | 'falling' | 'either' | 'level' | 'sampled level'  
**Default:** 'none'

**Initial condition source — Select source of initial condition**  
internal (default) | external

Select source of initial condition:

- **internal** — Get the initial conditions of the states from the **Initial condition** block parameter.
- **external** — Get the initial conditions of the states from an external block, via the IC input port.

**Dependencies**

Selecting **internal** enables the **Initial condition** parameter.

Selecting **external** disables the **Initial condition** parameter and enables the IC input port.

**Programmatic Use**  
**Block Parameter:** InitialConditionSource  
**Type:** character vector, string  
**Values:** 'internal' | 'external'  
**Default:** 'internal'

**Initial condition — Initial condition of states**  
0 (default) | scalar | vector | matrix

Specify initial condition of the block states. The minimum and maximum values are bound by the **Output minimum** and **Output maximum** block parameters.

**Tip** Simulink software does not allow the initial condition of this block to be `inf` or `NaN`.

**Dependencies**

To enable this parameter, set the **Initial condition source** to internal.
**Programmatic Use**

**Block Parameter:** InitialCondition  
**Type:** character vector, string  
**Values:** scalar | vector | matrix  
**Default:** '0'

**Initial condition setting — Select where to apply the initial condition**  
Auto (default) | Output | Compatibility

Select whether to apply the value of the **Initial condition** parameter to the block state or block output. The initial condition is also the reset value.

- **Auto** — Block chooses where to apply the **Initial condition** parameter.
  - If the block is in a non-triggered subsystem and **Integrator method** is set to an integration method, set initial conditions:
    
    \[ x(0) = IC \]
    
    At reset:
    
    \[ x(n) = IC \]
  
  - If the block is in a triggered or function-call subsystem and **Integrator method** is set to an integration method, set initial conditions as if **output** was selected.

- **Output** — Use this option when the block is in a triggered or a function-call subsystem and **Integrator method** is set to an integration method.

Set initial conditions:

\[ y(0) = IC \]

At reset:

\[ y(n) = IC \]

- **Compatibility** — This option is present to provide backward compatibility. You cannot select this option for Discrete-Time Integrator blocks in Simulink models but you can select it for Discrete-Time Integrator blocks in a library. Use this option to maintain compatibility with Simulink models created before R2014a.

Prior to R2014a, the option **Auto** was known as State only (most efficient). The option **Output** was known as State and output. The behavior of the block with the option **Compatibility** is as follows.
• If **Underspecified initialization detection** is set to **Classic**, the **Initial condition setting** parameter behaves as **Auto**.

• If **Underspecified initialization detection** is set to **Simplified**, the **Initial condition setting** parameter behaves as **Output**.

**Note** This parameter was named **Use initial condition as initial and reset value for** in Simulink before R2014a.

**Programmatic Use**
**Block Parameter:** InitialConditionSetting  
**Type:** character vector  
**Value:** 'Auto' | 'Output' | 'Compatibility'  
**Default:** 'Auto'

**Sample time (−1 for inherited) — Interval between samples**
-1 (default) | scalar | vector

Enter the discrete time interval between steps.

By default, the block uses a discrete sample time of 1. To set a different sample time, enter another discrete value, such as 0.1.

See “Specify Sample Time” for more information.

**Tips**
• Do not specify a sample time of 0. This value specifies a continuous sample time, which the Discrete-Time Integrator block does not support.

• Do not specify a sample time of inf or NaN because these values are not discrete.

• If you specify -1 to inherit the sample time from an upstream block, verify that the upstream block uses a discrete sample time. For example, the Discrete-Time Integrator block cannot inherit a sample time of 0.

**Programmatic Use**
**Block Parameter:** SampleTime  
**Type:** character vector  
**Values:** scalar | vector  
**Default:** ' -1'
**Limit output** — Limit block output values to specified range
off (default) | on

Limit the block's output to a value between the **Lower saturation limit** and **Upper saturation limit** parameters.

- Selecting this check box limits the block's output to a value between the **Lower saturation limit** and **Upper saturation limit** parameters.
- Clearing this check box does not limit the block's output values.

**Dependencies**

Selecting this parameter enables the **Lower saturation limit** and **Upper saturation limit** parameters.

**Programmatic Use**

**Block Parameter:** LimitOutput
**Type:** character vector
**Values:** 'off' | 'on'
**Default:** 'off'

**Upper saturation limit** — Upper limit for the integral
`inf` (default) | scalar | vector | matrix

Specify the upper limit for the integral as a scalar, vector, or matrix. You must specify a value between the **Output minimum** and **Output maximum** parameter values.

**Dependencies**

To enable this parameter, select the **Limit output** check box.

**Programmatic Use**

**Block Parameter:** UpperSaturationLimit
**Type:** character vector, string
**Values:** scalar | vector | matrix
**Default:** 'inf'

**Lower saturation limit** — Lower limit for the integral
`-inf` (default) | scalar | vector | matrix

Specify the lower limit for the integral as a scalar, vector, or matrix. You must specify a value between the **Output minimum** and **Output maximum** parameter values.
Dependencies

To enable this parameter, select the **Limit output** check box.

**Programmatic Use**

**Block Parameter:** LowerSaturationLimit

**Type:** character vector, string

**Values:** scalar | vector | matrix

**Default:** '-inf'

**Show saturation port — Enable saturation output port**

*off (default) | on*

Select this check box to add a saturation output port to the block. When you clear this check box, the block does not have a saturation output port.

**Dependencies**

Selecting this parameter enables a saturation output port.

**Programmatic Use**

**Block Parameter:** ShowSaturationPort

**Type:** character vector, string

**Values:** 'off' | 'on'

**Default:** 'off'

**Show state port — Enable state output port**

*off (default) | on*

Select this check box to add a state output port to the block. When you clear this check box, the block does not have a state output port.

**Dependencies**

Selecting this parameter enables a state output port.

**Programmatic Use**

**Block Parameter:** ShowStatePort

**Type:** character vector, string

**Values:** 'off' | 'on'

**Default:** 'off'

**Ignore limit and reset when linearizing — Treat block as not resettable**

*off (default) | on*
Select this check box to have Simulink linearization commands treat this block as not resettable and as having no limits on its output, regardless of the settings of the block reset and output limitation options.

**Tip** Ignoring the limit and resetting allows you to linearize a model around an operating point. This point may cause the integrator to reset or saturate.

**Programmatic Use**

**Block Parameter:** IgnoreLimit  
**Type:** character vector  
**Values:** 'off' | 'on'  
**Default:** 'off'

**Signal Attributes**

**Output minimum — Minimum output value for range checking**  
[] (default) | scalar

Lower value of the output range that Simulink checks.

Simulink uses the minimum to perform:

- Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters”) for some blocks.
- Simulation range checking (see “Signal Ranges” and “Enable Simulation Range Checking”).
- Automatic scaling of fixed-point data types.
- Optimization of the code that you generate from the model. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. For more information, see “Optimize using the specified minimum and maximum values” (Simulink Coder).

**Note** Output minimum does not saturate or clip the actual output signal. Use the Saturation block instead.
Programmatic Use
Block Parameter: OutMin
Type: character vector
Values: '[ ]'| scalar
Default: '[ ]'

Output maximum — Maximum output value for range checking
[] (default) | scalar

Upper value of the output range that Simulink checks.

Simulink uses the maximum value to perform:

- Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters”) for some blocks.
- Simulation range checking (see “Signal Ranges” and “Enable Simulation Range Checking”).
- Automatic scaling of fixed-point data types.
- Optimization of the code that you generate from the model. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. For more information, see “Optimize using the specified minimum and maximum values” (Simulink Coder).

**Note** Output maximum does not saturate or clip the actual output signal. Use the Saturation block instead.

Programmatic Use
Block Parameter: OutMax
Type: character vector
Values: '[ ]'| scalar
Default: '[ ]'

Data type — Output data type
Inherit: Inherit via internal rule (default)| Inherit: Inherit via back propagation | double | single | int8 | uint8 | int16 | uint16 | int32 | uint32 | int64 | uint64 | fixdt(1,16) | fixdt(1,16,0) | fixdt(1,16,2^0,0) | <data type expression>
Choose the data type for the output. The type can be inherited, specified directly, or expressed as a data type object such as Simulink.NumericType. For more information, see “Control Signal Data Types”.

When you select an inherited option, the block behaves as follows:

- **Inherit: Inherit via internal rule** — Simulink chooses a data type to balance numerical accuracy, performance, and generated code size, while taking into account the properties of the embedded target hardware. If you change the embedded target settings, the data type selected by the internal rule might change. For example, if the block multiplies an input of type int8 by a gain of int16 and ASIC/FPGA is specified as the targeted hardware type, the output data type is sfix24. If Unspecified (assume 32-bit Generic), i.e., a generic 32-bit microprocessor, is specified as the target hardware, the output data type is int32. If none of the word lengths provided by the target microprocessor can accommodate the output range, Simulink software displays an error in the Diagnostic Viewer.

It is not always possible for the software to optimize code efficiency and numerical accuracy at the same time. If the internal rule doesn’t meet your specific needs for numerical accuracy or performance, use one of the following options:

- Specify the output data type explicitly.
- Use the simple choice of **Inherit: Same as input**.
- Explicitly specify a default data type such as fixdt(1,32,16) and then use the Fixed-Point Tool to propose data types for your model. For more information, see fxptdlg.
- To specify your own inheritance rule, use **Inherit: Inherit via back propagation** and then use a Data Type Propagation block. Examples of how to use this block are available in the Signal Attributes library Data Type Propagation Examples block.
- **Inherit: Inherit via back propagation** — Use data type of the driving block.

**Programmatic Use**

**Block Parameter:** OutDataTypeStr  
**Type:** character vector  
**Values:** 'Inherit: Inherit via internal rule' | 'Inherit: Inherit via back propagation' | 'double' | 'single' | 'int8' | 'uint8' | 'int16' | 'uint16' | 'int32' | 'uint32' | 'int64' | 'uint64' | 'fixdt(1,16)' | 'fixdt(1,16,0)' | 'fixdt(1,16,2^0,0)' | '<data type expression>'  
**Default:** 'Inherit: Inherit via internal rule'
Lock output data type setting against changes by the fixed-point tools — Prevent fixed-point tools from overriding Output data type

off (default) | on

Select this parameter to prevent the fixed-point tools from overriding the Output data type you specify on the block. For more information, see “Use Lock Output Data Type Setting” (Fixed-Point Designer).

Programmatic Use

Block Parameter: LockScale

Type: character vector

Values: 'off' | 'on'

Default: 'off'

Integer rounding mode — Specify the rounding mode for fixed-point operations

Floor (default) | Ceiling | Convergent | Nearest | Round | Simplest | Zero

Choose one of these rounding modes.

Ceiling

Rounds both positive and negative numbers toward positive infinity. Equivalent to the MATLAB ceil function.

Convergent

Rounds number to the nearest representable value. If a tie occurs, rounds to the nearest even integer. Equivalent to the Fixed-Point Designer convergent function.

Floor

Rounds both positive and negative numbers toward negative infinity. Equivalent to the MATLAB floor function.

Nearest

Rounds number to the nearest representable value. If a tie occurs, rounds toward positive infinity. Equivalent to the Fixed-Point Designer nearest function.

Round

Rounds number to the nearest representable value. If a tie occurs, rounds positive numbers toward positive infinity and rounds negative numbers toward negative infinity. Equivalent to the Fixed-Point Designer round function.

Simplest

Automatically chooses between round toward floor and round toward zero to generate rounding code that is as efficient as possible.
Zero
Rounds number toward zero. Equivalent to the MATLAB `fix` function.

**Programmatic Use**

**Block Parameter:** RndMeth  
**Type:** character vector  
**Values:** 'Ceiling' | 'Convergent' | 'Floor' | 'Nearest' | 'Round' | 'Simplest' | 'Zero'  
**Default:** 'Floor'

**See Also**

For more information, see “Rounding” (Fixed-Point Designer).

**Saturate on integer overflow — Method of overflow action**

off (default) | on

Specify whether overflows saturate or wrap.

- off — Overflows wrap to the appropriate value that the data type can represent.
  
  For example, the number 130 does not fit in a signed 8-bit integer and wraps to -126.

- on — Overflows saturate to either the minimum or maximum value that the data type can represent.
  
  For example, an overflow associated with a signed 8-bit integer can saturate to -128 or 127.

**Tip**

- Consider selecting this check box when your model has a possible overflow and you want explicit saturation protection in the generated code.

- Consider clearing this check box when you want to optimize efficiency of your generated code.

  Clearing this check box also helps you to avoid overspecifying how a block handles out-of-range signals. For more information, see “Check for Signal Range Errors”.

- When you select this check box, saturation applies to every internal operation on the block, not just the output or result.

- In general, the code generation process can detect when overflow is not possible. In this case, the code generator does not produce saturation code.
Programmatic Use
Block Parameter: SaturateOnIntegerOverflow
Type: character vector
Values: 'off' | 'on'
Default: 'off'

State Attributes

State name — Unique name for block state
'' (default) | alphanumeric string

Use this parameter to assign a unique name to the block state. The default is ''. When this field is blank, no name is assigned. When using this parameter, remember these considerations:

• A valid identifier starts with an alphabetic or underscore character, followed by alphanumeric or underscore characters.
• The state name applies only to the selected block.

For more information, see “Apply Storage Classes to Individual Signal, State, and Parameter Data Elements” (Simulink Coder).

Dependencies

When you specify a value for State name and click Apply, you enable the State name must resolve to Simulink signal object parameter.

Programmatic Use
Parameter: StateName
Type: character vector
Values: unique name
Default: ''

State name must resolve to Simulink signal object — Require state names resolve to signal object
Off (default) | Boolean

Specify if requiring that state name resolve to Simulink signal objects or not. If selected, the software generates an error at run time if you specify a state name that does not match the name of a Simulink signal object.
Dependency

Enabled when you give the parameter **State name** a value and set the model configuration parameter **Signal resolution** to a value other than **None**.

Selecting this check box disables **Code generation storage class**.

**Programmatic Use**
*Block Parameter:* StateMustResolveToSignalObject
*Type:* character vector
*Values:* 'off' | 'on'
*Default:* 'off'

**Signal object class — Custom storage class package name**

Simulink.Signal (default)

Choose a custom storage class package by selecting a signal object class that the target package defines. For example, to apply custom storage classes from the built-in package **mpt**, select **mpt.Signal**. Unless you use an ERT-based code generation target with Embedded Coder, custom storage classes do not affect the generated code.

If the class that you want does not appear in the list, select **Customize class lists**. For instructions, see “Target Class Does Not Appear in List of Signal Object Classes” (Embedded Coder).

For information about storage classes, see “Apply Storage Classes to Individual Signal, State, and Parameter Data Elements” (Simulink Coder). For information about custom storage classes, see “Apply Built-In and Customized Storage Classes to Data Elements” (Embedded Coder).

**Programmatic Use**
*Block Parameter:* StateSignalObject
*Type:* character vector
*Values:* 'Simulink.Signal' | '<StorageClass.PackageName>'
*Default:* 'Simulink.Signal'

**Code generation storage class — Storage class for code generation**

Auto (default) | Model default | ExportedGlobal | ImportedExtern | ImportedExternPointer | Bitfield (Custom) | Volatile (Custom) | ExportToFile (Custom) | ImportFromFile (Custom) | FileScope (Custom) | Struct (Custom) | GetSet (Custom) | Reusable (Custom)
Select state storage class for code generation. If you do not need to interface to external code, select Auto.

For more information, see “Apply Storage Classes to Individual Signal, State, and Parameter Data Elements” (Simulink Coder) and “Apply Built-In and Customized Storage Classes to Data Elements” (Embedded Coder).

**Programmatic Use**

**Block Parameter:** `StateStorageClass`  
**Type:** character vector  
**Values:** 'Auto' | 'Model default' | 'ExportedGlobal' | 'ImportedExtern' | 'ImportedExternPointer' | 'Custom'  
**Default:** 'Auto'

**TypeQualifier — Storage type qualifier**  
'' (default) | const | volatile | ...

Specify a storage type qualifier such as const or volatile.

**Note** `TypeQualifier` will be removed in a future release. To apply storage type qualifiers to data, use custom storage classes and memory sections. Unless you use an ERT-based code generation target with Embedded Coder, custom storage classes and memory sections do not affect the generated code.

**Dependencies**

To enable this parameter, set **Code generation storage class** to ExportedGlobal, ImportedExtern, ImportedExternPointer, or Model default. This parameter is hidden unless you previously set its value.

**Programmatic Use**

**Block Parameter:** `RTWStateStorageTypeQualifier`  
**Type:** character vector  
**Values:** '' | 'const' | 'volatile' | ...

**Default:** ''
Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>double</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.
Depends on absolute time when used inside a triggered subsystem hierarchy.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.
HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture
This block has a single, default HDL architecture.
**HDL Block Properties**

<table>
<thead>
<tr>
<th>General</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ConstrainedOutputPipeline</strong></td>
</tr>
<tr>
<td><strong>InputPipeline</strong></td>
</tr>
<tr>
<td><strong>OutputPipeline</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Native Floating Point</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>HandleDenormals</strong></td>
</tr>
<tr>
<td><strong>LatencyStrategy</strong></td>
</tr>
<tr>
<td><strong>MantissaMultiplyStrategy</strong></td>
</tr>
</tbody>
</table>

**Restrictions**

- State ports are not supported for HDL code generation. Clear the **Show state port** option.
- External initial conditions are not supported for HDL code generation. Set **Initial condition source** to Internal.
• **External Reset** must be set to none, rising, or falling.
• Width of input and output signals must not exceed 32 bits.
• Continuous sample time is not supported. Use a discrete sample time for the block.

**PLC Code Generation**
Generate Structured Text code using Simulink® PLC Coder™.

**Fixed-Point Conversion**
Design and simulate fixed-point systems using Fixed-Point Designer™.

**See Also**
Integrator

**Topics**
“Apply Storage Classes to Individual Signal, State, and Parameter Data Elements” (Simulink Coder)
“Apply Storage Classes to Individual Signal, State, and Parameter Data Elements” (Simulink Coder)
“Apply Built-In and Customized Storage Classes to Data Elements” (Embedded Coder)

**Introduced before R2006a**
Discrete Transfer Fcn

Implement discrete transfer function

Library: Simulink / Discrete
HDL Coder / Discrete
HDL Coder / HDL Floating Point Operations

Description

The Discrete Transfer Fcn block implements the $z$-transform transfer function as follows:

$$H(z) = \frac{num(z)}{den(z)} = \frac{num_0 z^m + num_1 z^{m-1} + ... + num_m}{den_0 z^n + den_1 z^{n-1} + ... + den_n}$$

where $m+1$ and $n+1$ are the number of numerator and denominator coefficients, respectively. $num$ and $den$ contain the coefficients of the numerator and denominator in descending powers of $z$. $num$ can be a vector or matrix, while $den$ must be a vector. The order of the denominator must be greater than or equal to the order of the numerator.

Specify the coefficients of the numerator and denominator polynomials in descending powers of $z$. This block lets you use polynomials in $z$ to represent a discrete system, a method that control engineers typically use. Conversely, the Discrete Filter block lets you use polynomials in $z^{-1}$ (the delay operator) to represent a discrete system, a method that signal processing engineers typically use. The two methods are identical when the numerator and denominator polynomials have the same length.

The Discrete Transfer Fcn block applies the $z$-transform transfer function to each independent channel of the input. The Input processing parameter allows you to specify whether the block treats each column of the input as an individual channel (frame-based processing) or each element of the input as an individual channel (sample-based processing). To perform frame-based processing, you must have a DSP System Toolbox license.

Specifying Initial States

Use the Initial states parameter to specify initial filter states. To determine the number of initial states you must specify and how to specify them, use the following tables.
Frame-Based Processing

<table>
<thead>
<tr>
<th>Input</th>
<th>Number of Channels</th>
<th>Valid Initial States (Dialog Box)</th>
<th>Valid Initial States (Input Port)</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Column vector (K-by-1)</td>
<td>1</td>
<td>• Scalar</td>
<td>• Scalar</td>
</tr>
<tr>
<td>• Unoriented vector (K)</td>
<td></td>
<td>• Column vector (M-by-1)</td>
<td>• Column vector (M-by-1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Row vector (1-by-M)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Matrix (K-by-N)</td>
<td></td>
</tr>
<tr>
<td>• Row vector (1-by-N)</td>
<td>N</td>
<td>• Scalar</td>
<td>• Scalar</td>
</tr>
<tr>
<td>• Matrix (K-by-N)</td>
<td></td>
<td>• Column vector (M-by-1)</td>
<td>• Matrix (M-by-N)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Row vector (1-by-M)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Matrix (M-by-N)</td>
<td></td>
</tr>
</tbody>
</table>
Sample-Based Processing

<table>
<thead>
<tr>
<th>Input</th>
<th>Number of Channels</th>
<th>Valid Initial States (Dialog Box)</th>
<th>Valid Initial States (Input Port)</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Scalar</td>
<td>1</td>
<td>• Scalar</td>
<td>• Scalar</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Column vector (M-by-1)</td>
<td>• Column vector (M-by-1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Row vector (1-by-M)</td>
<td>• Row vector (1-by-M)</td>
</tr>
<tr>
<td>• Row vector (1-by-N)</td>
<td>N</td>
<td>• Scalar</td>
<td>• Scalar</td>
</tr>
<tr>
<td>• Column vector (N-by-1)</td>
<td></td>
<td>• Column vector (M-by-1)</td>
<td></td>
</tr>
<tr>
<td>• Unoriented vector (N)</td>
<td></td>
<td>• Row vector (1-by-M)</td>
<td></td>
</tr>
<tr>
<td>• Matrix (K-by-N)</td>
<td>K × N</td>
<td>• Scalar</td>
<td>• Scalar</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Column vector (M-by-1)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Row vector (1-by-M)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Matrix (M-by-(K×N))</td>
<td></td>
</tr>
</tbody>
</table>

When the **Initial states** is a scalar, the block initializes all filter states to the same scalar value. To initialize all states to zero, enter 0. When the **Initial states** is a vector or a matrix, each vector or matrix element specifies a unique initial state for a corresponding delay element in a corresponding channel:

- The vector length must equal the number of delay elements in the filter, \( M = \max(\text{number of zeros, number of poles}) \).
- The matrix must have the same number of rows as the number of delay elements in the filter, \( M = \max(\text{number of zeros, number of poles}) \). The matrix must also have one column for each channel of the input signal.

The following example shows the relationship between the initial filter output and the initial input and state. Given an initial input \( u_1 \), the first output \( y_1 \) is related to the initial state \([x_1, x_2]\) and initial input by as follows:
\[ y_1 = 4x_1 \]
\[ x_2 = \frac{1}{2}(u_1 - 3x_1) \]
Ports

Input

\( u \) — Input signal
scalar | vector | matrix

Input signal, specified as a scalar, vector, or matrix.
Data Types: single | double | int8 | int16 | int32 | fixed point

\( \text{Num} \) — Numerator coefficients
scalar | vector | matrix

Coefficients of the numerator polynomial specified as a vector or matrix in descending powers of \( z \). Use a row vector to specify the coefficients for a single numerator polynomial. Use a matrix to specify coefficients for multiple filters to be applied to the same input. Each matrix row represents a set of filter taps. The order of the denominator must be greater than or equal to the order of the numerator.

Dependencies
To enable this port, set Numerator Source to Input port.

Numerator and denominator coefficients must have the same complexity. They can have different word lengths and fraction lengths.
Data Types: single | double | int8 | int16 | int32 | fixed point

\( \text{Den} \) — Denominator coefficients
scalar | vector | matrix

Coefficients of the denominator polynomial specified as a vector in descending powers of \( z \). Use a row vector to specify the coefficients for a single denominator polynomial. Use a matrix to specify coefficients for multiple filters to be applied to the same input. Each matrix row represents a set of filter taps. The order of the denominator must be greater than or equal to the order of the numerator.

Dependencies
To enable this port, set Denominator Source to Input port.
Numerator and denominator coefficients must have the same complexity. They can have different word lengths and fraction lengths.

Data Types: single | double | int8 | int16 | int32 | fixed point

**External reset — External reset signal**

scalar

External reset signal, specified as a scalar. When the specified trigger event occurs, the block resets the states to their initial conditions.

**Tip** The icon for this port changes based on the value of the External reset parameter.

**Dependencies**

To enable this port, set **External reset** to Rising, Falling, Either, Level, or Level hold.

**Limitations**

The reset signal must be a scalar of type single, double, Boolean, or integer. Fixed-point data types, except for ufix1, are not supported.

Data Types: single | double | Boolean | int8 | int16 | int32 | fixed point

**x0 — Initial states**

scalar | vector | matrix

Initial states, specified as a scalar, vector, or matrix. For more information about specifying states, see “Specifying Initial States” on page 1-638. States are complex when either the input or the coefficients are complex.

**Dependencies**

To enable this port, set **Initial states Source** to Input port.

Data Types: single | double | int8 | int16 | int32 | fixed point

**Output**

**Port_1 — Output signal**

scalar | vector | matrix
Output signal specified as a scalar, vector, or matrix.

Data Types: single | double | int8 | int16 | int32 | fixed point

Parameters

Main

Numerator Source — Source of numerator coefficients
Dialog (default) | Input port

Specify the source of the numerator coefficients as Dialog or Input port.

Programmatic Use
Block Parameter: NumeratorSource
Type: character vector
Values: 'Dialog' | 'Input port'
Default: 'Dialog'

Numerator Value — Numerator coefficients
[1] (default) | scalar | vector | matrix

Numerator coefficients of the discrete transfer function. To specify the coefficients, set the Source to Dialog. Then, enter the coefficients in Value as descending powers of z. Use a row vector to specify the coefficients for a single numerator polynomial. Use a matrix to specify coefficients for multiple filters to be applied to the same input. Each matrix row represents a set of filter taps.

Dependencies

To enable this parameter, set the Numerator Source to Dialog.

Programmatic Use
Block Parameter: Numerator
Type: character vector
Values: scalar | vector | matrix
Default: '[1]'

Denominator Source — Source of denominator coefficients
Dialog (default) | Input port

Specify the source of the denominator coefficients as Dialog or Input port.
Programmatic Use

Block Parameter: DenominatorSource
Type: character vector
Values: 'Dialog' | 'Input port'
Default: 'Dialog'

Denominator Value — Denominator coefficients
[1 0.5] (default) | scalar | vector | matrix

Denominator coefficients of the discrete transfer function. To specify the coefficients, set the Source to Dialog. Then, enter the coefficients in Value as descending powers of z. Use a row vector to specify the coefficients for a single denominator polynomial. Use a matrix to specify coefficients for multiple filters to be applied to the same input. Each matrix row represents a set of filter taps.

Dependencies
To enable this parameter, set the Denominator Source to Dialog.

Programmatic Use

Block Parameter: Denominator
Type: character vector
Values: scalar | vector | matrix
Default: '[1 0.5]'

Initial states Source — Source of initial states
Dialog (default) | Input port

Specify the source of the initial states as Dialog or Input port.

Programmatic Use

Block Parameter: InitialStatesSource
Type: character vector
Values: 'Dialog' | 'Input port'
Default: 'Dialog'

Initial states Value — Initial filter states
0 (default) | scalar | vector | matrix

Specify the initial filter states as a scalar, vector, or matrix. To learn how to specify initial states, see “Specifying Initial States” on page 1-638.
Dependencies

To enable this parameter, set **Initial states Source** to **Dialog**.

**Programmatic Use**

**Block Parameter:** InitialStates  
**Type:** character vector  
**Values:** scalar | vector | matrix  
**Default:** '0'

**External reset — External state reset**

None (default) | Rising | Falling | Either | Level | Level hold

Specify the trigger event to use to reset the states to the initial conditions.

<table>
<thead>
<tr>
<th>Reset Mode</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>No reset</td>
</tr>
<tr>
<td>Rising</td>
<td>Reset on a rising edge</td>
</tr>
<tr>
<td>Falling</td>
<td>Reset on a falling edge</td>
</tr>
<tr>
<td>Either</td>
<td>Reset on either a rising or falling edge</td>
</tr>
<tr>
<td>Level</td>
<td>Reset in either of these cases:</td>
</tr>
<tr>
<td></td>
<td>• When the reset signal is nonzero at the current time step</td>
</tr>
<tr>
<td></td>
<td>• When the reset signal value changes from nonzero at the previous time</td>
</tr>
<tr>
<td></td>
<td>step to zero at the current time step</td>
</tr>
<tr>
<td>Level hold</td>
<td>Reset when the reset signal is nonzero at the current time step</td>
</tr>
</tbody>
</table>

**Programmatic Use**

**Block Parameter:** ExternalReset  
**Type:** character vector  
**Values:** 'None' | 'Rising' | 'Falling' | 'Either' | 'Level' | 'Level hold'  
**Default:** 'None'

**Input processing — Sample- or frame-based processing**

Elements as channels (sample based) (default) | Columns as channels (frame based)
Specify whether the block performs sample- or frame-based processing.

- **Elements as channels (sample based)** — Process each element of the input as an independent channel.
- **Columns as channels (frame based)** — Process each column of the input as an independent channel.

**Note** Frame-based processing requires a DSP System Toolbox license.

For more information, see “Sample- and Frame-Based Concepts” (DSP System Toolbox).

**Programmatic Use**

**Block Parameter:** InputProcessing

**Type:** character vector

**Values:** 'Elements as channels (sample based)' | 'Columns as channels (frame based)'

**Default:** 'Elements as channels (sample based)'

**Optimize by skipping divide by leading denominator coefficient (a0)**

— **Skip divide by a0**

**off** (default) | **on**

Select when the leading denominator coefficient, \( a_0 \), equals 1. This parameter optimizes your code.

When you select this check box, the block does not perform a divide-by-\( a_0 \) either in simulation or in the generated code. An error occurs if \( a_0 \) is not equal to one.

When you clear this check box, the block is fully tunable during simulation, and performs a divide-by-\( a_0 \) in both simulation and code generation.

**Programmatic Use**

**Block Parameter:** a0EqualsOne

**Type:** character vector

**Values:** 'off' | 'on'

**Default:** 'off'

**Sample time (-1 for inherited)** — **Interval between samples**

-1 (default) | scalar | vector
Specify the time interval between samples. To inherit the sample time, set this parameter to -1. For more information, see “Specify Sample Time”.

**Programmatic Use**

**Block Parameter:** `SampleTime`

**Type:** character vector

**Values:** scalar | vector

**Default:** '-1'

---

## Data Types

### State — State data type

**Inherit:** Same as input (default) | int8 | int16 | int32 | int64 | fixdt(1,16,0) | <data type expression>

Specify the state data type. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Same as input`
- A built-in integer, for example, `int8`
- A data type object, for example, a `Simulink.NumericType` object
- An expression that evaluates to a data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button to display the **Data Type Assistant**, which helps you set the data type attributes. For more information, see “Specify Data Types Using Data Type Assistant”.

### Numerator coefficients — Numerator coefficient data type

**Inherit:** Inherit via internal rule (default) | int8 | int16 | int32 | int64 | fixdt(1,16) | fixdt(1,16,0) | <data type expression>

Specify the numerator coefficient data type. You can set it to:

- A rule that inherits a data type, for example, `Inherit: Inherit via internal rule`
- A built-in integer, for example, `int8`
- A data type object, for example, a `Simulink.NumericType` object
- An expression that evaluates to a data type, for example, `fixdt(1,16,0)`
Click the Show data type assistant button to display the Data Type Assistant, which helps you set the data type attributes. For more information, see “Specify Data Types Using Data Type Assistant”.

**Programmatic Use**

**Block Parameter:** NumCoeffDataTypeStr  
**Type:** character vector  
**Values:** 'Inherit: Inherit via internal rule' | 'int8' | 'int16' | 'int32' | 'int64' | 'fixdt(1,16)' | 'fixdt(1,16,0)' | '<data type expression>'  
**Default:** 'Inherit: Inherit via internal rule'

**Numerator coefficient minimum — Minimum value of numerator coefficients**

[] (default) | scalar

Specify the minimum value that a numerator coefficient can have. The default value is [] (unspecified). Simulink software uses this value to perform:

- Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters”)
- Automatic scaling of fixed-point data types

**Programmatic Use**

**Block Parameter:** NumCoeffMin  
**Type:** character vector  
**Values:** scalar  
**Default:** '[]'

**Numerator coefficient maximum — Maximum value of numerator coefficients**

[] (default) | scalar

Specify the maximum value that a numerator coefficient can have. The default value is [] (unspecified). Simulink software uses this value to perform:

- Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters”)
- Automatic scaling of fixed-point data types

**Programmatic Use**

**Block Parameter:** NumCoeffMax
**Type:** character vector

**Values:** scalar

**Default:** ' []'

### Numerator product output — Numerator product output data type

Inherit: Inherit via internal rule (default) | Inherit: Same as input | int8 | int16 | int32 | int64 | fixdt(1,16,0) | <data type expression>

Specify the product output data type for the numerator coefficients. You can set it to:

- A rule that inherits a data type, for example, Inherit: Inherit via internal rule
- A built-in data type, for example, int8
- A data type object, for example, a Simulink.NumericType object
- An expression that evaluates to a data type, for example, fixdt(1,16,0)

Click the **Show data type assistant** button to display the **Data Type Assistant**, which helps you set the data type attributes. For more information, see “Specify Data Types Using Data Type Assistant”.

### Programmatic Use

**Block Parameter:** NumProductDataTypeStr

**Type:** character vector

**Values:** 'Inherit: Inherit via internal rule' | 'Inherit: Same as input' | 'int8' | 'int16' | 'int32' | 'int64' | 'fixdt(1,16,0)' | '<data type expression>'

**Default:** 'Inherit: Inherit via internal rule'

### Numerator accumulator — Numerator accumulator data type

Inherit: Inherit via internal rule (default) | Inherit: Same as input | Inherit: Same as product output | int8 | int16 | int32 | int64 | fixdt(1,16,0) | <data type expression>

Specify the accumulator data type for the numerator coefficients. You can set it to:

- A rule that inherits a data type, for example, Inherit: Inherit via internal rule
- A built-in data type, for example, int8
- A data type object, for example, a Simulink.NumericType object
• An expression that evaluates to a data type, for example, fixdt(1,16,0)

Click the **Show data type assistant** button to display the **Data Type Assistant**, which helps you set the data type attributes. For more information, see “Specify Data Types Using Data Type Assistant”.

**Programmatic Use**

**Block Parameter:** NumAccumDataTypeStr  
**Type:** character vector  
**Values:** 'Inherit: Inherit via internal rule' | 'Inherit: Same as input' | 'Inherit: Same as product output' | 'int8' | 'int16' | 'int32' | 'int64' | 'fixdt(1,16)' | 'fixdt(1,16,0)' | '<data type expression>'  
**Default:** 'Inherit: Inherit via internal rule'

**Denominator coefficients — Denominator coefficient data type**

Inherit: Inherit via internal rule (default) | int8 | int16 | int32 | int64 | fixdt(1,16) | fixdt(1,16,0) | <data type expression>

Specify the denominator coefficient data type. You can set it to:

• A rule that inherits a data type, for example, Inherit: Inherit via internal rule
• A built-in integer, for example, int8
• A data type object, for example, a Simulink.NumericType object
• An expression that evaluates to a data type, for example, fixdt(1,16,0)

Click the **Show data type assistant** button to display the **Data Type Assistant**, which helps you set the data type attributes. For more information, see “Specify Data Types Using Data Type Assistant”.

**Programmatic Use**

**Block Parameter:** DenCoeffDataTypeStr  
**Type:** character vector  
**Values:** 'Inherit: Inherit via internal rule' | 'Inherit: Same as input' | 'int8' | 'int16' | 'int32' | 'int64' | 'fixdt(1,16)' | 'fixdt(1,16,0)' | '<data type expression>'  
**Default:** 'Inherit: Same wordlength as input'
Denominator coefficient minimum — Minimum value of denominator coefficients
[] (default) | scalar

Specify the minimum value that a denominator coefficient can have. The default value is [] (unspecified). Simulink software uses this value to perform:

• Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters”)
• Automatic scaling of fixed-point data types

Programmatic Use
Block Parameter: DenCoeffMin
Type: character vector
Values: scalar
Default: '[]'

Denominator coefficient maximum — Maximum value of denominator coefficients
[] (default) | scalar

Specify the maximum value that a denominator coefficient can have. The default value is [] (unspecified). Simulink software uses this value to perform:

• Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters”)
• Automatic scaling of fixed-point data types

Programmatic Use
Block Parameter: DenCoeffMax
Type: character vector
Values: scalar
Default: '[]'

Denominator product output — Denominator product output data type
Inherit: Inherit via internal rule (default) | Inherit: Same as input | int8 | int16 | int32 | int64 | fixdt(1,16,0) | <data type expression>

Specify the product output data type for the denominator coefficients. You can set it to:

• A rule that inherits a data type, for example, **Inherit: Inherit via internal rule**
• A built-in data type, for example, int8
• A data type object, for example, a Simulink.NumericType object
• An expression that evaluates to a data type, for example, fixdt(1,16,0)

Click the **Show data type assistant** button to display the **Data Type Assistant**, which helps you set the data type attributes. For more information, see “Specify Data Types Using Data Type Assistant”.

**Programmatic Use**

**Block Parameter:** DenProductDataTypeStr  
**Type:** character vector  
**Values:** 'Inherit: Inherit via internal rule' | 'Inherit: Same as input' | 'int8' | 'int16' | 'int32' | 'int64' | 'fixdt(1,16,0)' | '<data type expression>'  
**Default:** 'Inherit: Inherit via internal rule'

**Denominator accumulator — Denominator accumulator data type**

Inherit: Inherit via internal rule (default) | Inherit: Same as input | Inherit: Same as product output | int8 | int16 | int32 | int64 | fixdt(1,16,0) | <data type expression>

Specify the accumulator data type for the denominator coefficients. You can set it to:

• A rule that inherits a data type, for example, Inherit: Inherit via internal rule
• A built-in data type, for example, int8
• A data type object, for example, a Simulink.NumericType object
• An expression that evaluates to a data type, for example, fixdt(1,16,0)

Click the **Show data type assistant** button to display the **Data Type Assistant**, which helps you set the data type attributes. For more information, see “Specify Data Types Using Data Type Assistant”.

**Programmatic Use**

**Block Parameter:** DenAccumDataTypeStr  
**Type:** character vector
Values: 'Inherit: Inherit via internal rule' | 'Inherit: Same as input' | 'Inherit: Same as product output' | 'int8' | 'int16' | 'int32' | 'int64' | 'fixdt(1,16,0)' | '<data type expression>'
Default: 'Inherit: Inherit via internal rule'

Output — Output data type

Specify the output data type. You can set it to:

- A rule that inherits a data type, for example, Inherit: Inherit via internal rule
- A built-in data type, for example, int8
- A data type object, for example, a Simulink.NumericType object
- An expression that evaluates to a data type, for example, fixdt(1,16,0)

Click the Show data type assistant button to display the Data Type Assistant, which helps you set the data type attributes. For more information, see “Specify Data Types Using Data Type Assistant”.

Programmatic Use

Block Parameter: OutDataTypeStr
Type: character vector
Values: 'Inherit: Inherit via internal rule' | 'Inherit: Same as input' | 'int8' | 'int16' | 'int32' | 'int64' | 'fixdt(1,16)' | 'fixdt(1,16,0)' | '<data type expression>'
Default: 'Inherit: Inherit via internal rule'

Output minimum — Minimum value of output

Specify the minimum value that the block can output. The default value is [] (unspecified). Simulink uses this value to perform:

- Simulation range checking (see “Signal Ranges”)
- Automatic scaling of fixed-point data types
Programmatic Use
Block Parameter: OutMin
Type: character vector
Values: scalar
Default: ‘[]’

Output maximum — Maximum value of output
[] (default) | scalar

Specify the maximum value that the block can output. The default value is [] (unspecified). Simulink uses this value to perform:

• Simulation range checking (see “Signal Ranges”)
• Automatic scaling of fixed-point data types

Programmatic Use
Block Parameter: OutMax
Type: character vector
Values: scalar
Default: ‘[]’

Lock data type settings against changes by the fixed-point tools — Prevent fixed-point tools from overriding data types
off (default) | on

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on this block. For more information, see “Lock the Output Data Type Setting” (Fixed-Point Designer).

Programmatic Use
Block Parameter: LockScale
Type: character vector
Values: 'off' | 'on'
Default: 'off'

Integer rounding mode — Rounding mode for fixed-point operations
Floor (default) | Ceiling | Convergent | Nearest | Round | Simplest | Zero

Specify the rounding mode for fixed-point operations. For more information, see “Rounding” (Fixed-Point Designer).
Block parameters always round to the nearest representable value. To control the rounding of a block parameter, enter an expression using a MATLAB rounding function into the mask field.

**Programmatic Use**
**Block Parameter:** RndMeth  
**Type:** character vector  
**Values:** 'Ceiling' | 'Convergent' | 'Floor' | 'Nearest' | 'Round' | 'Simplest' | 'Zero'  
**Default:** 'Floor'

**Saturate on integer overflow — Method of overflow action**
off (default) | on

Specify whether overflows saturate or wrap.

<table>
<thead>
<tr>
<th>Action</th>
<th>Rationale</th>
<th>Impact on Overflows</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Select this check box (on).</td>
<td>Your model has possible overflow, and you want explicit saturation protection in the generated code.</td>
<td>Overflows saturate to either the minimum or maximum value that the data type can represent.</td>
<td>The maximum value that the int8 (signed, 8-bit integer) data type can represent is 127. Any block operation result greater than this maximum value causes overflow of the 8-bit integer. With the check box selected, the block output saturates at 127. Similarly, the block output saturates at a minimum output value of -128.</td>
</tr>
<tr>
<td>Action</td>
<td>Rationale</td>
<td>Impact on Overflows</td>
<td>Example</td>
</tr>
<tr>
<td>--------</td>
<td>-----------</td>
<td>---------------------</td>
<td>---------</td>
</tr>
<tr>
<td>Do not select this check box (off).</td>
<td>You want to optimize efficiency of your generated code. You want to avoid overspecifying how a block handles out-of-range signals. For more information, see “Check for Signal Range Errors”.</td>
<td>Overflows wrap to the appropriate value that is representable by the data type.</td>
<td>The maximum value that the int8 (signed, 8-bit integer) data type can represent is 127. Any block operation result greater than this maximum value causes overflow of the 8-bit integer. With the check box cleared, the software interprets the overflow-causing value as int8, which can produce an unintended result. For example, a block result of 130 (binary 1000 0010) expressed as int8, is -126.</td>
</tr>
</tbody>
</table>

When you select this check box, saturation applies to every internal operation on the block, not just the output, or result. Usually, the code generation process can detect when overflow is not possible. In this case, the code generator does not produce saturation code.

**Programmatic Use**

**Block Parameter:** SaturateOnIntegerOverflow  
**Type:** character vector  
**Values:** 'off' | 'on'  
**Default:** 'off'

**State Attributes**

**State name — Unique name for block state**  
' ' (default) | alphanumeric string

Use this parameter to assign a unique name to the block state. The default is ' '. When this field is blank, no name is assigned. When using this parameter, remember these considerations:

- A valid identifier starts with an alphabetic or underscore character, followed by alphanumeric or underscore characters.
The state name applies only to the selected block.

This parameter enables **State name must resolve to Simulink signal object** when you click **Apply**.

For more information, see “Apply Storage Classes to Individual Signal, State, and Parameter Data Elements” (Simulink Coder).

**Programmatic Use**
**Block Parameter:** StateName
**Type:** character vector
**Values:** unique name
**Default:** ‘’

**State name must resolve to Simulink signal object — Require state name resolve to a signal object**
* off (default) | on

Select this check box to require that the state name resolves to a Simulink signal object.

**Dependencies**
To enable this parameter, specify a value for **State name**. This parameter appears only if you set the model configuration parameter **Signal resolution** to a value other than None.

Selecting this check box disables **Code generation storage class**.

**Programmatic Use**
**Block Parameter:** StateMustResolveToSignalObject
**Type:** character vector
**Values:** 'off' | 'on'
**Default:** 'off'

**Signal object class — Custom storage class package name**
* Simulink.Signal (default) | <StorageClass.PackageName>

Choose a custom storage class package by selecting a signal object class that the target package defines. For example, to apply custom storage classes from the built-in package mpt, select mpt.Signal. Unless you use an ERT-based code generation target with Embedded Coder, custom storage classes do not affect the generated code.

If the class that you want does not appear in the list, select **Customize class lists**. For instructions, see “Target Class Does Not Appear in List of Signal Object Classes” (Embedded Coder).
For information about storage classes, see “Apply Storage Classes to Individual Signal, State, and Parameter Data Elements” (Simulink Coder). For information about custom storage classes, see “Apply Built-In and Customized Storage Classes to Data Elements” (Embedded Coder).

**Programmatic Use**

**Block Parameter:** StateSignalObject

**Type:** character vector

**Values:** 'Simulink.Signal' | '<StorageClass.PackageName>'

**Default:** 'Simulink.Signal'

**Code generation storage class — State storage class for code generation**

Auto (default) | Model default | ExportedGlobal | ImportedExtern | ImportedExternPointer | BitField (Custom) | Model default | ExportToFile (Custom) | ImportFromFile (Custom) | FileScope (Custom) | AutoScope (Custom) | Struct (Custom) | GetSet (Custom) | Reusable (Custom)

Select state storage class for code generation.

- Auto is the appropriate storage class for states that you do not need to interface to external code.
- StorageClass applies the storage class or custom storage class that you select from the list. For information about storage classes, see “Apply Storage Classes to Individual Signal, State, and Parameter Data Elements” (Simulink Coder). For information about custom storage classes, see “Apply Built-In and Customized Storage Classes to Data Elements” (Embedded Coder).

Use **Signal object class** to select custom storage classes from a package other than Simulink.

**Dependencies**

To enable this parameter, specify a value for **State name**.

**Programmatic Use**

**Block Parameter:** StateStorageClass

**Type:** character vector

**Values:** 'Auto' | 'Model default' | 'ExportedGlobal' | 'ImportedExtern' | 'ImportedExternPointer' | 'Custom' | ...

**Default:** 'Auto'

**TypeQualifier — Storage type qualifier**

' ' (default) | const | volatile | ...
Specify a storage type qualifier such as `const` or `volatile`.

**Note** `TypeQualifier` will be removed in a future release. To apply storage type qualifiers to data, use custom storage classes and memory sections. Unless you use an ERT-based code generation target with Embedded Coder, custom storage classes and memory sections do not affect the generated code.

During simulation, the block uses the following values:

- The initial value of the signal object to which the state name is resolved
- Minimum and Maximum values of the signal object

For more information, see “Data Objects”.

**Dependencies**

To enable this parameter, set `Code generation storage class` to `ExportedGlobal`, `ImportedExtern`, `ImportedExternPointer`, or `Model default`. This parameter is hidden unless you previously set its value.

**Programmatic Use**

**Block Parameter:** `RTWStateStorageTypeQualifier`  
**Type:** character vector  
**Values:** `' ' | 'const' | 'volatile' | ...`  
**Default:** `' '``

### Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>double</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Direct Feedthrough</strong></td>
<td>no</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Multidimensional Signals</strong></td>
<td>no</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Variable-Size Signals</strong></td>
<td>no</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Zero-Crossing Detection | no
---|---
a. This block only supports signed fixed-point data types.

## Extended Capabilities

### C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

### HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

### HDL Architecture
This block has a single, default HDL architecture.

### HDL Block Properties

<table>
<thead>
<tr>
<th>General</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConstMultiplierOptimization</td>
<td>Canonical signed digit (CSD) or factored CSD optimization. The default is none. See also “ConstMultiplierOptimization” (HDL Coder).</td>
</tr>
<tr>
<td>ConstrainedOutputPipeline</td>
<td>Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. For more details, see “ConstrainedOutputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td>InputPipeline</td>
<td>Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “InputPipeline” (HDL Coder).</td>
</tr>
</tbody>
</table>
### General

| **OutputPipeline** | Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “OutputPipeline” (HDL Coder). |

### Native Floating Point

| **HandleDenormals** | Specify whether you want HDL Coder to insert additional logic to handle denormal numbers in your design. Denormal numbers are numbers that have magnitudes less than the smallest floating-point number that can be represented without leading zeros in the mantissa. The default is inherit. See also “HandleDenormals” (HDL Coder). |

| **LatencyStrategy** | Specify whether to map the blocks in your design to inherit, Max, Min, or Zero for the floating-point operator. The default is inherit. See also “LatencyStrategy” (HDL Coder). |

| **MantissaMultiplyStrategy** | Specify how to implement the mantissa multiplication operation during code generation. By using different settings, you can control the DSP usage on the target FPGA device. The default is inherit. See also “MantissaMultiplyStrategy” (HDL Coder). |

### Restrictions

- You must use the **Inherit: Inherit via internal rule** option for data type propagation only if the input data type is double.
- Frame, matrix, and vector input data types are not supported.
- The leading denominator coefficient (a0) must be 1 or -1.

The Discrete Transfer Fcn block is excluded from the following optimizations:

- Resource sharing
- Distributed pipelining

### PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.
Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

This block only supports signed fixed-point data types.

See Also
Discrete Filter | Transfer Fcn

Topics
“Working with States” on page 13-25
“Apply Storage Classes to Individual Signal, State, and Parameter Data Elements” (Simulink Coder)

Introduced before R2006a
Discrete Zero-Pole

Model system defined by zeros and poles of discrete transfer function

Library: Simulink / Discrete

Description

The Discrete Zero-Pole block models a discrete system defined by the zeros, poles, and gain of a \( z \)-domain transfer function. This block assumes that the transfer function has the following form:

\[
H(z) = \frac{KZ(z)}{P(z)} = K\frac{(z - Z_1)(z - Z_2)\ldots(z - Z_m)}{(z - P_1)(z - P_2)\ldots(z - P_n)},
\]

where \( Z \) represents the zeros vector, \( P \) the poles vector, and \( K \) the gain. The number of poles must be greater than or equal to the number of zeros \( (n \geq m) \). If the poles and zeros are complex, they must be complex conjugate pairs.

The block displays the transfer function depending on how the parameters are specified. See Zero-Pole for more information.

Modeling a Single-Output System

For a single-output system, the input and the output of the block are scalar time-domain signals. To model this system:

1. Enter a vector for the zeros of the transfer function in the Zeros field.
2. Enter a vector for the poles of the transfer function in the Poles field.
3. Enter a 1-by-1 vector for the gain of the transfer function in the Gain field.

Modeling a Multiple-Output System

For a multiple-output system, the block input is a scalar and the output is a vector, where each element is an output of the system. To model this system:
1 Enter a matrix of zeros in the Zeros field. Each column of this matrix contains the zeros of a transfer function that relates the system input to one of the outputs.

2 Enter a vector for the poles common to all transfer functions of the system in the Poles field.

3 Enter a vector of gains in the Gain field. Each element is the gain of the corresponding transfer function in Zeros.

Each element of the output vector corresponds to a column in Zeros.

Ports

Input

Port_1 — Input signal
scalar

Input signal specified as a real-valued scalar.

Data Types: single | double

Output

Port_1 — Model of discrete system
scalar | vector

Model of system as defined by zeros, poles, and gain of discrete transfer function. The width of the output is equal to the number of columns in the Zeros matrix, or one if Zeros is a vector.

Data Types: single | double
Parameters

Main

Zeros — Matrix of zeros

Specify the vector or matrix of zeros. The number of zeros must be less than or equal to the number of poles. If the poles and zeros are complex, they must be complex conjugate pairs.

- For a single-output system, enter a vector for the zeros of the transfer function.
- For a multiple-output system, enter a matrix. Each column of the matrix contains the zeros of a transfer function that relates the system input to one of the outputs.

Programmatic Use

Block Parameter: Zeros
Type: character vector
Values: vector
Default: ' [1] '

Poles — Vector of poles

Specify the vector of poles. The number of poles must be greater than or equal to the number of zeros. If the poles and zeros are complex, they must be complex conjugate pairs.

- For a single-output system, enter a vector for the poles of the transfer function.
- For a multiple-output system, enter a vector for the poles common to all transfer functions of the system.

Programmatic Use

Block Parameter: Poles
Type: character vector
Values: vector
Default: ' [0 0.5] '

Gain — Gain value

1 (default) | scalar | vector
Specify vector of gain values.

- For a single-output system, enter a scalar or 1-by-1 vector for the gain of the transfer function.
- For a multiple-output system, enter a vector of gains. Each element is the gain of the corresponding transfer function in **Zeros**.

**Programmatic Use**

**Block Parameter:** Gain  
**Type:** character vector  
**Values:** scalar | vector  
**Default:** '1'

**Sample time (-1 for inherited) — Interval between samples**

-1 | scalar | vector

Specify the time interval between samples. For more information, see Specifying Sample Time.

**Programmatic Use**

**Block Parameter:** SampleTime  
**Type:** character vector  
**Values:** scalar | vector  
**Default:** '-1'

**State Attributes**

**State name — Unique name for block state**

' ' (default) | alphanumeric string

Use this parameter to assign a unique name to the block state. The default is ' '. When this field is blank, no name is assigned. When using this parameter, remember these considerations:

- A valid identifier starts with an alphabetic or underscore character, followed by alphanumeric or underscore characters.
- The state name applies only to the selected block.

This parameter enables **State name must resolve to Simulink signal object** when you click **Apply**.
For more information, see “Apply Storage Classes to Individual Signal, State, and Parameter Data Elements” (Simulink Coder).

**Programmatic Use**

**Block Parameter:** StateName  
**Type:** character vector  
**Values:** unique name  
**Default:** ''

*State name must resolve to Simulink signal object — Require state name resolve to a signal object*

*off* (default) | *on*

Select this check box to require that the state name resolves to a Simulink signal object.

**Dependencies**

To enable this parameter, specify a value for **State name**. This parameter appears only if you set the model configuration parameter **Signal resolution** to a value other than **None**.

Selecting this check box disables **Code generation storage class**.

**Programmatic Use**

**Block Parameter:** StateMustResolveToSignalObject  
**Type:** character vector  
**Values:** 'off' | 'on'  
**Default:** 'off'

*Signal object class — Custom storage class package name*

*Simulink.Signal* (default) | <StorageClass.PackageName>

Choose a custom storage class package by selecting a signal object class that the target package defines. For example, to apply custom storage classes from the built-in package *mpt*, select *mpt.Signal*. Unless you use an ERT-based code generation target with Embedded Coder, custom storage classes do not affect the generated code.

If the class that you want does not appear in the list, select **Customize class lists**. For instructions, see “Target Class Does Not Appear in List of Signal Object Classes” (Embedded Coder).

For information about storage classes, see “Apply Storage Classes to Individual Signal, State, and Parameter Data Elements” (Simulink Coder). For information about custom storage classes, see “Apply Built-In and Customized Storage Classes to Data Elements” (Embedded Coder).
Programmatic Use
Block Parameter: StateSignalObject
Type: character vector
Values: 'Simulink.Signal' | '<StorageClass.PackageName>'
Default: 'Simulink.Signal'

Code generation storage class — State storage class for code generation
Auto (default) | Model default | ExportedGlobal | ImportedExtern |
ImportedExternPointer | BitField (Custom) | Model default | ExportToFile (Custom) |
ImportFromFile (Custom) | FileScope (Custom) | AutoScope (Custom) |
Struct (Custom) | GetSet (Custom) | Reusable (Custom)

Select state storage class for code generation.

- Auto is the appropriate storage class for states that you do not need to interface to external code.
- StorageClass applies the storage class or custom storage class that you select from the list. For information about storage classes, see “Apply Storage Classes to Individual Signal, State, and Parameter Data Elements” (Simulink Coder). For information about custom storage classes, see “Apply Built-In and Customized Storage Classes to Data Elements” (Embedded Coder).

Use Signal object class to select custom storage classes from a package other than Simulink.

Dependencies
To enable this parameter, specify a value for State name.

Programmatic Use
Block Parameter: StateStorageClass
Type: character vector
Values: 'Auto' | 'Model default' | 'ExportedGlobal' | 'ImportedExtern' |
'ImportedExternPointer' | 'Custom' | ...
Default: 'Auto'

TypeQualifier — Storage type qualifier
'' (default) | const | volatile | ...

Specify a storage type qualifier such as const or volatile.
**Note** TypeQualifier will be removed in a future release. To apply storage type qualifiers to data, use custom storage classes and memory sections. Unless you use an ERT-based code generation target with Embedded Coder, custom storage classes and memory sections do not affect the generated code.

During simulation, the block uses the following values:

- The initial value of the signal object to which the state name is resolved
- Minimum and Maximum values of the signal object

For more information, see “Data Objects”.

**Dependencies**

To enable this parameter, set **Code generation storage class** to ExportedGlobal, ImportedExtern, ImportedExternPointer, or Model default. This parameter is hidden unless you previously set its value.

**Programmatic Use**

**Block Parameter:** RTWStateStorageTypeQualifier  
**Type:** character vector  
**Values:** '' | 'const' | 'volatile' | ...  
**Default:** ''

**Block Characteristics**

<table>
<thead>
<tr>
<th>Data Types</th>
<th>double</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Direct Feedthrough</strong></td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td><strong>Multidimensional Signals</strong></td>
<td>no</td>
<td></td>
</tr>
<tr>
<td><strong>Variable-Size Signals</strong></td>
<td>no</td>
<td></td>
</tr>
<tr>
<td><strong>Zero-Crossing Detection</strong></td>
<td>no</td>
<td></td>
</tr>
</tbody>
</table>
Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

Generated code relies on memcpy or memset functions (string.h) under certain conditions.

See Also
Discrete Transfer Fcn | Zero-Pole

Topics
“Apply Storage Classes to Individual Signal, State, and Parameter Data Elements” (Simulink Coder)
“Apply Built-In and Customized Storage Classes to Data Elements” (Embedded Coder)
“Data Objects”

Introduced before R2006a
Display

Display signal value during simulation
Library: Simulink / Dashboard

Description

The Display block connects to a signal in your model and displays its value during simulation. You can configure the appearance and format for the Display block to make intuitive sense for the value it displays. You can edit the parameters of the Display block during simulation. Use the Display block with other Dashboard blocks to build an interactive dashboard of controls and indicators for your model.

Connecting Dashboard Blocks

Dashboard blocks do not use ports to connect to signals. To connect Dashboard blocks to signals in your model, use connect mode. Connect mode facilitates the process of connecting Dashboard blocks in your model, especially when you want to connect multiple blocks at once. If you only want to connect a single Dashboard block, you can also use the Connection table in the block dialog box to make the connection.

Tip To facilitate understanding and debugging your model, you can modify Dashboard block connections in your model during normal and accelerator mode simulations.

To enter connect mode, click the Connect button that appears above your unconnected Dashboard block when you pause on it.
In connect mode, when you select one or more signals or blocks, a list of parameters and signals available for connection appears. Select a signal or parameter from the list to connect the selected Dashboard block. To connect another Dashboard block, pause on the block and click the **Connect** button above it. Then, make a selection of signals and blocks in your model, and choose a signal or parameter to connect.
When you finish connecting the Dashboard blocks in your model, click the Exit button in the upper-right of the canvas to exit connect mode.

**Tip** You can hide the message shown on unconnected blocks using the `set_param` function and the `ShowInitialText` block parameter.

**Limitations**

- Dashboard blocks can only connect to real scalar signals.
- You cannot use the **Connection** table to connect a Dashboard block to a block that is commented out. When you connect a Dashboard block to a commented block using connect mode, the Dashboard block does not display the connected value until the you uncomment the block.
- Dashboard blocks cannot connect to signals inside referenced models.
- When you simulate a model hierarchy, Dashboard blocks inside referenced models do not update.
- Dashboard blocks do not support rapid accelerator simulation.
Some signals are not available during accelerator mode simulations. To view such signals on a Dashboard block, mark the signal for logging.

**Parameters**

**Connection — Select a signal to connect and display**

signal connection options

Select the signal to connect using the **Connection** table. Populate the **Connection** table by selecting signals of interest in your model. Select the radio button next to the signal you want to display. Click **Apply** to connect the signal. To facilitate understanding and debugging your model, you can connect Dashboard blocks to signals in your model during simulation.

**Programmatic Use**

**Block Parameter:** Binding  
**Type:** Simulink.HMI.SignalSpecification  
**Default:** []

**Format — Format for displaying numerical values**

short (default) | long | shortE | longE | ...

Format for displaying numerical values. You can specify any format supported by the `format` command:

- **short** — Scaled fixed-decimal format with four digits after the decimal point
- **long** — Scaled fixed-decimal format with fifteen digits after the decimal point for `double` values and seven digits after the decimal point for `single` values
- **shortE** — Scientific notation format with four digits
- **longE** — Scientific notation format with fifteen digits after the decimal point for `double` values and seven digits after the decimal point for `single` values
- **shortG** — Data takes the more compact format between fixed-decimal or scientific notation, with a total of five digits
- **longG** — Data takes the more compact format between fixed-decimal or scientific notation, with a total of fifteen digits for `double` values and seven digits for `single` values
- **shortEng** — Engineering notation where the exponent is a multiple of 3, with 4 digits after the decimal point
• **longEng** — Engineering notation where the exponent is a multiple of 3, with 15 significant digits
• **+** — Positive/negative format. +, -, and blank characters are displayed for positive, negative, and zero values, respectively
• **bank** — Currency format with 2 digits after the decimal point
• **hex** — Hexadecimal representation
• **rat** — Ratio

### Programmatic Use

**Block Parameter:** Format  
**Type:** character array  
**Values:** 'short' | 'long' | 'shortE' | 'longE' | 'shortG' | 'longG' | 'shortEng' | 'longEng' | 'bank' | '+' | 'hex' | 'rat'  
**Default:** 'short'

### Alignment — Text alignment in block

Center (default) | Left | Right
Text alignment in the Display block.

**Programmatic Use**

**Block Parameter:** Alignment  
**Type:** character array  
**Values:** 'Left' | 'Center' | 'Right'  
**Default:** 'Center'

**Label — Block label position**

Hide (default) | Bottom | Top

Position of the block label. When the block is connected to an element in the model, the label is the name of the connected element.

**Programmatic Use**

**Block Parameter:** LabelPosition  
**Type:** character vector  
**Values:** 'Hide' | 'Bottom' | 'Top'  
**Default:** 'Hide'

**Opacity — Specify block opacity**

1 (default) | scalar

Opacity of the block, specified as a scalar value between 0 and 1.

Example: 0.5

**Programmatic Use**

**Block Parameter:** Opacity  
**Type:** scalar  
**Default:** 1

**Foreground Color — Specify foreground color**

1 (default) | scalar

Foreground color for the block, including the text. You can select a color from a palette of standard colors, or you can specify a custom color.

**Programmatic Use**

Specify the **ForegroundColor** parameter for the block as a 1-by-3 [r g b] vector with values between 0 and 1.  
**Block Parameter:** ForegroundColor
Type: [r g b] vector
Default: [0 0 0]

Background Color — Specify background color
1 (default) | scalar

Block background color. You can select a color from a palette of standard colors, or you can specify a custom color.

Programmatic Use

Specify the BackgroundColor parameter for the block as a 1-by-3 [r g b] vector with values between 0 and 1.
Block Parameter: BackgroundColor
Type: [r g b] vector
Default: [1 1 1]

Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>double</th>
<th>single</th>
<th>Boolean</th>
<th>base integer</th>
<th>fixed point</th>
<th>enumerated</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.
Dashboard blocks are ignored for code generation.

**See Also**
Dashboard Scope | Gauge | Lamp | MultiStateImage

**Topics**
“Tune and Visualize Your Model with Dashboard Blocks”
“Decide How to Visualize Simulation Data”

**Introduced in R2017b**
Display

Show value of input

Library: Simulink / Sinks

Description

The Display block shows the value of the input data. You can specify the frequency of the display. For numeric input data, you can also specify the format of display.

If the block input is an array, you can resize the block vertically or horizontally to show more than just the first element. If the block input is a vector, the block sequentially adds display fields from left to right and top to bottom. The block displays as many values as possible. A black triangle indicates that the block is not displaying all input array elements.

The Display block shows the first 200 elements of a vector signal and the first 20 rows and 10 columns of a matrix signal.

Note If you specify a numeric display format that is not large enough to display all the digits of a value, the displayed values may lose precision. The result depends on your computer hardware and operating system.

Display Abbreviations

The following abbreviations appear on the Display block to help you identify the format of the value.
When You See... | The Value That Appears Is...
---|---
(SI) | The stored integer value

**Note** (SI) does not appear when the signal is of an integer data type.

<table>
<thead>
<tr>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hex</td>
<td>In hexadecimal format</td>
</tr>
<tr>
<td>bin</td>
<td>In binary format</td>
</tr>
<tr>
<td>oct</td>
<td>In octal format</td>
</tr>
</tbody>
</table>

### Displaying Strings

When working with strings, the Display block displays:

- Strings with double quotes.
- Special characters such as newline are shown as escaped sequences, for example '\n'.
- Non-displayable characters as escaped octal number, for example '\201'.

If the incoming signal is of type string, the **Numeric display format** parameter selection does not affect the display of the string.

### Ports

#### Input

**Port_1 — Input data**

**scalar | vector**

Input data to display.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated
Parameters

Numeric display format — Format to display numeric input data
short (default) | long | short_e | long_e | bank | hex (Stored Integer) | binary (Stored Integer) | decimal (Stored Integer) | octal (Stored Integer)

Specify the format of the numeric data that appears.

<table>
<thead>
<tr>
<th>If You Select...</th>
<th>The Block Displays...</th>
</tr>
</thead>
<tbody>
<tr>
<td>short</td>
<td>A 5-digit scaled value with fixed decimal point</td>
</tr>
<tr>
<td>long</td>
<td>A 15-digit scaled value with fixed decimal point</td>
</tr>
<tr>
<td>short_e</td>
<td>A 5-digit value with a floating decimal point</td>
</tr>
<tr>
<td>long_e</td>
<td>A 16-digit value with a floating decimal point</td>
</tr>
<tr>
<td>bank</td>
<td>A value in fixed dollars and cents format (but with no $ or commas)</td>
</tr>
<tr>
<td>hex (Stored Integer)</td>
<td>The stored integer value of a fixed-point input in hexadecimal format</td>
</tr>
<tr>
<td>binary (Stored Integer)</td>
<td>The stored integer value of a fixed-point input in binary format</td>
</tr>
<tr>
<td>decimal (Stored Integer)</td>
<td>The stored integer value of a fixed-point input in decimal format</td>
</tr>
<tr>
<td>octal (Stored Integer)</td>
<td>The stored integer value of a fixed-point input in octal format</td>
</tr>
</tbody>
</table>

If the numeric input to a Display block has an enumerated data type (see “Simulink Enumerations” and “Define Simulink Enumerations”):

- The block displays enumerated values, not the values of underlying integers.
- Setting **Numeric display format** to any of the **Stored Integer** settings causes an error.

If the incoming signal is of type string, the selection of the **Numeric display format** parameter does not affect the display of the string.
Programmatic Use
Block Parameter: Format
Type: character vector
Values: 'short' | 'long' | 'short_e' | 'long_e' | 'bank' | 'hex (Stored Integer)' | 'binary (Stored Integer)' | 'decimal (Stored Integer)' | 'octal (Stored Integer)'
Default: 'short'

Decimation — Display rate
1 (default) | integer

Specify how often to display data.

The amount of data that appears and the time steps at which the data appears depend on the Decimation block parameter and the SampleTime property.

• The Decimation parameter enables you to display data at every \( n \)th sample, where \( n \) is the decimation factor. The default decimation, 1, displays data at every time step.

\[
\text{Note} \quad \text{The Display block updates its display at the initial time, even when the Decimation value is greater than one.}
\]

• The SampleTime property, which you can set with set_param, enables you to specify a sampling interval at which to display points. This property is useful when you are using a variable-step solver where the interval between time steps is not the same. The default sample time, -1, causes the block to ignore the sampling interval when determining the points to display.

\[
\text{Note} \quad \text{If the block inherits a sample time of } \infty, \text{ the Decimation parameter is ignored.}
\]

Programmatic Use
Block Parameter: Decimation
Type: character vector
Values: '1' | integer
Default: '1'

Floating display — Floating display
off (default) | on
To use the block as a floating display, select the **Floating display** check box. The block input port disappears and the block displays the value of the signal on a selected line.

If you select **Floating display**:

- Turn off signal storage reuse for your model. See “Signal storage reuse” (Simulink Coder) for more information.
- Do not connect a multidimensional signal to a floating display.

**Programmatic Use**

**Block Parameter**: Floating

**Type**: character vector

**Values**: 'on' | 'off'

**Default**: 'on'

---

### Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>double</th>
<th>single</th>
<th>Boolean</th>
<th>base integer</th>
<th>fixed point</th>
<th>enumerated</th>
<th>string</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>Yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>Yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

### Extended Capabilities

**C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.
HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

This block can be used for simulation visibility in subsystems that generate HDL code, but is not included in the hardware implementation.

PLC Code Generation
Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

See Also
Scope | To File | To Workspace

Introduced before R2006a
Divide

Divide one input by another

**Library:**
- Simulink / Math Operations
- HDL Coder / HDL Floating Point Operations
- HDL Coder / Math Operations

![Divide block icon](image)

**Description**

The Divide block outputs the result of dividing its first input by its second. The inputs can be scalars, a scalar and a nonscalar, or two nonscalars that have the same dimensions. The Divide block is functionally a Product block that has two block parameter values preset:

- **Multiplication** — Element-wise(.*)
- **Number of Inputs** — */

Setting nondefault values for either of those parameters can change a Divide block to be functionally equivalent to a Product block or a Product of Elements block.

**Ports**

**Input**

X — **Input signal to multiply**
scalar | vector | matrix | N-D array

Input signal to be multiplied with other inputs.

**Dependencies**

To enable one or more X ports, specify one or more * characters for the **Number of inputs** parameter.
Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point

÷ — Input signal to divide or invert
scalar | vector | matrix | N-D array

Input signal for division or inversion operations.

Dependencies

To enable one or more ÷ ports, specify one or more / characters for the Number of inputs parameter.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point

Port_1 — First input to multiply or divide
scalar | vector | matrix | N-D array

First input to multiply or divide, provided as a scalar, vector, matrix, or N-D array.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point

Port_N — Nth input to multiply or divide
scalar | vector | matrix | N-D array

Nth input to multiply or divide, provided as a scalar, vector, matrix, or N-D array.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point

Output

Port_1 — Output computed by multiplying, dividing, or inverting inputs
scalar | vector | matrix | N-D array

Output computed by multiplying, dividing, or inverting inputs.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point
Parameters

Main

Number of inputs — Control number of inputs and type of operation
*/ (default) | positive integer scalar | * or / for each input port

Control two properties of the block:

- The number of input ports on the block
- Whether each input is multiplied or divided into the output

When you specify:

- 1 or * or /
  
  The block has one input port. In element-wise mode, the block processes the input as described for the Product of Elements block. In matrix mode, if the parameter value is 1 or *, the block outputs the input value. If the value is /, the input must be a square matrix (including a scalar as a degenerate case) and the block outputs the matrix inverse. See “Element-Wise Mode” on page 1-1519 and “Matrix Mode” on page 1-1520 for more information.

- Integer value > 1

  The block has the number of inputs given by the integer value. The inputs are multiplied together in element-wise mode or matrix mode, as specified by the Multiplication parameter. See “Element-Wise Mode” on page 1-1519 and “Matrix Mode” on page 1-1520 for more information.

- Unquoted string of two or more * and / characters

  The block has the number of inputs given by the length of the character vector. Each input that corresponds to a * character is multiplied into the output. Each input that corresponds to a / character is divided into the output. The operations occur in element-wise mode or matrix mode, as specified by the Multiplication parameter. See “Element-Wise Mode” on page 1-1519 and “Matrix Mode” on page 1-1520 for more information.

Programmatic Use
Block Parameter: Inputs
**Type**: character vector  
**Values**: '2' | '*' | '**' | '*/' | '*/*' | ...  
**Default**: '*/*'

**Multiplication — Element-wise (*) or Matrix (*) multiplication**  
Element-wise(.*)(default) | Matrix(*)

Specify whether the block performs Element-wise(.*') or Matrix(*) multiplication.

**Programmatic Use**  
**Block Parameter**: Multiplication  
**Type**: character vector  
**Values**: 'Element-wise(.*')' | 'Matrix(*')'  
**Default**: 'Element-wise(.*')'

**Multiply over — All dimensions or specified dimension**  
All dimensions (default) | Specified dimension

Specify the dimension to multiply over as All dimensions, or Specified dimension. When you select Specified dimension, you can specify the **Dimension** as 1 or 2.

**Dependencies**

To enable this parameter, set **Number of inputs** to * and **Multiplication** to Element-wise (.*').

**Programmatic Use**  
**Block Parameter**: CollapseMode  
**Type**: character vector  
**Values**: 'All dimensions' | 'Specified dimension'  
**Default**: 'All dimensions'

**Dimension — Dimension to multiply over**  
1 (default) | 2 | ... | N

Specify the dimension to multiply over as an integer less than or equal to the number of dimensions of the input signal.

**Dependencies**

To enable this parameter, set:

- **Number of inputs** to *
• **Multiplication** to Element-wise (*)
• **Multiply over** to Specified dimension

**Programmatic Use**

**Block Parameter:** CollapseDim  
**Type:** character vector  
**Values:** '1' | '2' | ...  
**Default:** '1'

**Sample time — Specify sample time as a value other than -1**
-1 (default) | scalar | vector

Specify the sample time as a value other than -1. For more information, see “Specify Sample Time”.

**Dependencies**

This parameter is not visible unless it is explicitly set to a value other than -1. To learn more, see “Blocks for Which Sample Time Is Not Recommended”.

**Programmatic Use**

**Block Parameter:** SampleTime  
**Type:** character vector  
**Values:** scalar or vector  
**Default:** '-1'

**Signal Attributes**

**Require all inputs to have the same data type — Require that all inputs have the same data type**
off (default) | on

Specify if input signals must all have the same data type. If you enable this parameter, then an error occurs during simulation if the input signal types are different.

**Programmatic Use**

**Block Parameter:** InputSameDT  
**Type:** character vector  
**Values:** 'off' | 'on'  
**Default:** 'off'

**Output minimum — Minimum output value for range checking**
[] (default) | scalar
Lower value of the output range that Simulink checks.

Simulink uses the minimum to perform:

- Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters”) for some blocks.
- Simulation range checking (see “Signal Ranges” and “Enable Simulation Range Checking”).
- Automatic scaling of fixed-point data types.
- Optimization of the code that you generate from the model. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. For more information, see “Optimize using the specified minimum and maximum values” (Simulink Coder).

**Note** Output minimum does not saturate or clip the actual output signal. Use the Saturation block instead.

**Programmatic Use**

**Block Parameter:** OutMin  
**Type:** character vector  
**Values:** '[]' | scalar  
**Default:** '[]'

**Output maximum — Maximum output value for range checking**

[ ] (default) | scalar

Upper value of the output range that Simulink checks.

Simulink uses the maximum value to perform:

- Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters”) for some blocks.
- Simulation range checking (see “Signal Ranges” and “Enable Simulation Range Checking”).
- Automatic scaling of fixed-point data types.
- Optimization of the code that you generate from the model. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. For more information, see “Optimize using the specified minimum and maximum values” (Simulink Coder).
Note Output maximum does not saturate or clip the actual output signal. Use the Saturation block instead.

Programmatic Use
Block Parameter: OutMax
Type: character vector
Values: '[]' scalar
Default: '[]'

Output data type — Specify the output data type
Inherit: Inherit via internal rule (default)| Inherit: Inherit via back propagation | Inherit: Same as first input | double | single | int8 | uint8 | int16 | uint16 | int32 | uint32 | int64 | uint64 | fixdt(1,16) | fixdt(1,16,0) | fixdt(1,16,2^0,0) | <data type expression>

Choose the data type for the output. The type can be inherited, specified directly, or expressed as a data type object such as Simulink.NumericType. For more information, see “Control Signal Data Types”.

When you select an inherited option, the block behaves as follows:

- Inherit: Inherit via internal rule — Simulink chooses a data type to balance numerical accuracy, performance, and generated code size, while taking into account the properties of the embedded target hardware. If you change the embedded target settings, the data type selected by the internal rule might change. For example, if the block multiplies an input of type int8 by a gain of int16 and ASIC/FPGA is specified as the targeted hardware type, the output data type is sfix24. If Unspecified (assume 32-bit Generic), in other words, a generic 32-bit microprocessor, is specified as the target hardware, the output data type is int32. If none of the word lengths provided by the target microprocessor can accommodate the output range, Simulink software displays an error in the Diagnostic Viewer.

It is not always possible for the software to optimize code efficiency and numerical accuracy at the same time. If the internal rule doesn’t meet your specific needs for numerical accuracy or performance, use one of the following options:

- Specify the output data type explicitly.
- Use the simple choice of Inherit: Same as input.
- Explicitly specify a default data type such as fixdt(1,32,16) and then use the Fixed-Point Tool to propose data types for your model. For more information, see fxptdlg.
• To specify your own inheritance rule, use Inherit: Inherit via back propagation and then use a Data Type Propagation block. Examples of how to use this block are available in the Signal Attributes library Data Type Propagation Examples block.

• Inherit: Inherit via back propagation — Use data type of the driving block.

• Inherit: Same as first input — Use data type of first input signal.

**Programmatic Use**
Block Parameter: OutDataTypeStr
Type: character vector
Values: 'Inherit: Inherit via internal rule' | 'Inherit: Same as first input' | 'Inherit: Inherit via back propagation' | 'double' | 'single' | 'int8' | 'uint8' | 'int16' | 'uint16' | 'int32' | 'uint32' | 'int64' | 'uint64' | 'fixdt(1,16)' | 'fixdt(1,16,0)' | 'fixdt(1,16,2^0,0)' | '<data type expression>'
Default: 'Inherit: Inherit via internal rule'

**Lock output data type setting against changes by the fixed-point tools — Prevent fixed-point tools from overriding Output data type**
off (default) | on

Select this parameter to prevent the fixed-point tools from overriding the Output data type you specify on the block. For more information, see “Use Lock Output Data Type Setting” (Fixed-Point Designer).

**Programmatic Use**
Block Parameter: LockScale
Type: character vector
Values: 'off' | 'on'
Default: 'off'

**Integer rounding mode — Rounding mode for fixed-point operations**
Floor (default) | Ceiling | Convergent | Nearest | Round | Simplest | Zero

Select the rounding mode for fixed-point operations. You can select:

Ceiling
Rounds positive and negative numbers toward positive infinity. Equivalent to the MATLAB ceil function.
Convergent
Rounds number to the nearest representable value. If a tie occurs, rounds to the nearest even integer. Equivalent to the Fixed-Point Designer `convergent` function.

Floor
Rounds positive and negative numbers toward negative infinity. Equivalent to the `MATLAB floor` function.

Nearest
Rounds number to the nearest representable value. If a tie occurs, rounds toward positive infinity. Equivalent to the Fixed-Point Designer `nearest` function.

Round
Rounds number to the nearest representable value. If a tie occurs, rounds positive numbers toward positive infinity and rounds negative numbers toward negative infinity. Equivalent to the Fixed-Point Designer `round` function.

Simplest
Chooses between rounding toward floor and rounding toward zero to generate rounding code that is as efficient as possible.

Zero
Rounds number toward zero. Equivalent to the `MATLAB fix` function.

For more information, see “Rounding” (Fixed-Point Designer).

Block parameters always round to the nearest representable value. To control the rounding of a block parameter, enter an expression using a `MATLAB` rounding function into the mask field.

**Programmatic Use**

**Block Parameter:** RndMeth  
**Type:** character vector  
**Values:** 'Ceiling' | 'Convergent' | 'Floor' | 'Nearest' | 'Round' | 'Simplest' | 'Zero'  
**Default:** 'Floor'

**Saturate on integer overflow — Method of overflow action**
off (default) | on

Specify whether overflows saturate or wrap.
<table>
<thead>
<tr>
<th>Action</th>
<th>Rationale</th>
<th>Impact on Overflows</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Select this check box (on).</td>
<td>Your model has possible overflow, and you want explicit saturation protection in the generated code.</td>
<td>Overflows saturate to either the minimum or maximum value that the data type can represent.</td>
<td>The maximum value that the int8 (signed, 8-bit integer) data type can represent is 127. Any block operation result greater than this maximum value causes overflow of the 8-bit integer. With the check box selected, the block output saturates at 127. Similarly, the block output saturates at a minimum output value of -128.</td>
</tr>
<tr>
<td>Do not select this check box (off).</td>
<td>You want to optimize efficiency of your generated code.</td>
<td>Overflows wrap to the appropriate value that is representable by the data type.</td>
<td>The maximum value that the int8 (signed, 8-bit integer) data type can represent is 127. Any block operation result greater than this maximum value causes overflow of the 8-bit integer. With the check box cleared, the software interprets the overflow-causing value as int8, which can produce an unintended result. For example, a block result of 130 (binary 1000 0010) expressed as int8, is -126.</td>
</tr>
</tbody>
</table>

When you select this check box, saturation applies to every internal operation on the block, not just the output, or result. Usually, the code generation process can detect when overflow is not possible. In this case, the code generator does not produce saturation code.
Programmatic Use

**Block Parameter:** SaturateOnIntegerOverflow

**Type:** character vector

**Values:** 'off' | 'on'

**Default:** 'off'

### Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>double</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Extended Capabilities

#### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

#### Expected Differences Between Simulation and Code Generation

These conditions may yield different results between simulation and the generated code:

- The Divide block inputs contain a NaN or inf value
- The Divide block generates NaN or inf during execution

This difference is due to the nonfinite NaN or inf values. In such cases, inspect your model configuration and eliminate the conditions that produce NaN or inf.
Code Optimizations

The Simulink Coder build process provides efficient code for matrix inverse and division operations. This table describes the benefits and when each benefit is available.

<table>
<thead>
<tr>
<th>Benefit</th>
<th>Small Matrices (2-by-2 to 5-by-5)</th>
<th>Medium Matrices (6-by-6 to 20-by-20)</th>
<th>Large Matrices (larger than 20-by-20)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Faster code execution time, compared to R2011a and earlier releases</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Reduced ROM and RAM usage, compared to R2011a and earlier releases</td>
<td>Yes, for real values</td>
<td>Yes, for real values</td>
<td>Yes, for real values</td>
</tr>
<tr>
<td>Reuse of variables</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Dead code elimination</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Constant folding</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Expression folding</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Consistency with MATLAB Coder results</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

For blocks that have three or more inputs of different dimensions, the code might include an extra buffer to store temporary variables for intermediate results.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.
**Note** When you deploy the generated HDL code onto the target hardware, make sure that you set the **signed integer division rounds to** parameter in the **Hardware Implementation** pane of the Configuration Parameters dialog box to **Zero** or **Simplest**.

To perform an HDL-optimized divide operation, connect a **Product block** to a **Divide block** in **reciprocal mode**.

**HDL Architecture**

**Default Mode**

The **Divide block** is the same as a **Product block** with **Number of Inputs** set to */.*.

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear default</td>
<td>None</td>
<td>Generate a divide (/) operator in the HDL code.</td>
</tr>
<tr>
<td>ShiftAdd</td>
<td>UsePipelines</td>
<td>Perform divide operations on fixed-point types by using a non-restoring division algorithm that performs multiple shift and add operations to compute the quotient. This architecture provides improved accuracy compared to the Newton-Raphson approximation method. When you use this architecture, to achieve a higher maximum clock frequency on the target FPGA device, leave the <strong>UsePipelines</strong> HDL block property to <strong>on</strong>.</td>
</tr>
</tbody>
</table>

**Reciprocal Mode**

When **Number of Inputs** is set to /, the **Divide block** is in **reciprocal mode**.

This block has multi-cycle implementations that introduce additional latency in the generated code. To see the added latency, view the generated model or validation model. See “Generated Model and Validation Model” (HDL Coder).
In reciprocal mode, the Divide block has the HDL block implementations described in the following table.

<table>
<thead>
<tr>
<th>Architectures</th>
<th>Parameter(s)</th>
<th>Additional cycles of latency</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>default Linear</td>
<td>None</td>
<td>0</td>
<td>When you compute a reciprocal, use the HDL divide (/) operator to implement the division.</td>
</tr>
<tr>
<td>ReciprocalRsqrtBasedNewton</td>
<td>Iterations</td>
<td>Signed input: Iterations + 5</td>
<td>Use the iterative Newton method. Select this option to optimize area.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Unsigned input: Iterations + 3</td>
<td>The default value for Iterations is 3.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>The recommended value for Iterations is between 2 and 10. If Iterations is outside the recommended range, HDL Coder displays a message.</td>
</tr>
<tr>
<td>Architectures</td>
<td>Parameters</td>
<td>Additional cycles of latency</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------------------------------</td>
<td>------------</td>
<td>------------------------------</td>
<td>-------------</td>
</tr>
</tbody>
</table>
| ReciprocalRsqrtBasedNewtonSingleRate              | Iterations | Signed input: (Iterations * 4) + 8  
Unsigned input: (Iterations * 4) + 6 | Use the single rate pipelined Newton method. Select this option to optimize speed, or if you want a single rate implementation.  
The default value for Iterations is 3.  
The recommended value for Iterations is between 2 and 10. If Iterations is outside the recommended range, the coder displays a message. |
<table>
<thead>
<tr>
<th>Architectures</th>
<th>Parameter(s)</th>
<th>Additional cycles of latency</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ShiftAdd</td>
<td>UsePipelines</td>
<td>Signed input: (Input wordlength + 4) Unsigned input: (Input wordlength + 4)</td>
<td>Perform reciprocal operation on a fixed-point input by using a non-restoring division algorithm that performs multiple shift and add operations to compute the reciprocal. This architecture provides improved accuracy compared to the Newton-Raphson approximation method. When you use this architecture, to achieve a higher maximum clock frequency on the target FPGA device, leave the UsePipelines HDL block property to on.</td>
</tr>
</tbody>
</table>

The Newton-Raphson iterative method:

ReciprocalRsqrtBasedNewton and ReciprocalRsqrtBasedNewtonSingleRate implement the Newton-Raphson method with:

**HDL Block Properties**

<table>
<thead>
<tr>
<th>General</th>
<th>Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. For more details, see “ConstrainedOutputPipeline” (HDL Coder).</th>
</tr>
</thead>
</table>
### General

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ConstrainedOutputPipeline</strong></td>
<td>Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. For more details, see “ConstrainedOutputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td><strong>DSPStyle</strong></td>
<td>Synthesis attributes for multiplier mapping. The default is none. See also “DSPStyle” (HDL Coder). Use this property with:</td>
</tr>
<tr>
<td></td>
<td>• Product block</td>
</tr>
<tr>
<td></td>
<td>• Divide and Reciprocal blocks with Linear architecture</td>
</tr>
<tr>
<td><strong>InputPipeline</strong></td>
<td>Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “InputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td><strong>OutputPipeline</strong></td>
<td>Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “OutputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td><strong>UsePipelines</strong></td>
<td>HDL property that specifies whether to use a pipelined implementation for fixed-point division and reciprocal operations. This implementation provides a higher maximum clock frequency on the target FPGA device. See also “UsePipelines” (HDL Coder).</td>
</tr>
</tbody>
</table>

### Native Floating Point

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>HandleDenormals</strong></td>
<td>Specify whether you want HDL Coder to insert additional logic to handle denormal numbers in your design. Denormal numbers are numbers that have magnitudes less than the smallest floating-point number that can be represented without leading zeros in the mantissa. The default is inherit. See also “HandleDenormals” (HDL Coder).</td>
</tr>
<tr>
<td><strong>LatencyStrategy</strong></td>
<td>Specify whether to map the blocks in your design to inherit, Max, Min, Zero, or Custom for the floating-point operator. The default is inherit. See also “LatencyStrategy” (HDL Coder).</td>
</tr>
</tbody>
</table>
### Native Floating Point

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NFPCustomLatency</td>
<td>To specify a value, set LatencyStrategy to Custom. HDL Coder adds latency equal to the value that you specify for the NFPCustomLatency setting. See also “NFPCustomLatency” (HDL Coder).</td>
</tr>
<tr>
<td>MantissaMultiplyStrategy</td>
<td>Specify how to implement the mantissa multiplication operation during code generation. By using different settings, you can control the DSP usage on the target FPGA device. The default is inherit. See also “MantissaMultiplyStrategy” (HDL Coder).</td>
</tr>
<tr>
<td>DivisionAlgorithm</td>
<td>Specify whether to use the Radix-2 or Radix-4 algorithm to perform the floating-point division. The Radix-2 mode offers a trade-off between latency and frequency. The Radix-4 mode offers a trade-off between latency and resource usage. For more information, see “DivisionAlgorithm” (HDL Coder).</td>
</tr>
</tbody>
</table>

### Complex Data Support

This block does not support code generation for division with complex signals.

### Restrictions

When you use the Divide block in reciprocal mode, the following restrictions apply:

- When you use fixed-point types, the input and output must be scalar. To use vector inputs, specify the Math architecture and input a floating-point value.
- Only the Zero rounding mode is supported.
- You must select the Saturate on integer overflow option on the block.

For the Divide block, only the Zero and Simplest rounding modes are supported.

### PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

### Fixed-Point Conversion

Design and simulate fixed-point systems using Fixed-Point Designer™.
See Also
Dot Product | Product | Product of Elements

Introduced before R2006a
DocBlock

Create text that documents model and save text with model
Library: Simulink / Model-Wide Utilities
HDL Coder / Model-Wide Utilities

Description

The DocBlock allows you to create and edit text that documents a model, and save that text with the model. Double-clicking an instance of the block creates a temporary file containing the text associated with this block and opens the file in an editor. Use the editor to modify the text and save the file. Simulink software stores the contents of the saved file in the model file.

The DocBlock supports HTML, Rich Text Format (RTF), and ASCII text document types. The default editors for these different document types are:

- **HTML** — Microsoft® Word (if available). Otherwise, the DocBlock opens HTML documents using the editor specified on the **Editor/Debugger Preferences** pane of the Preferences dialog box.
- **RTF** — Microsoft Word (if available). Otherwise, the DocBlock opens RTF documents using the editor specified on the **Editor/Debugger Preferences** pane of the Preferences dialog box.
- **Text** — The DocBlock opens text documents using the editor specified on the **Editor/Debugger Preferences** pane of the Preferences dialog box.

Use the `docblock` command to change the default editors.

**Tip** To edit the block parameters of the DocBlock, right-click the block icon and select **Mask > Mask Parameters**....
Parameters

Code generation template symbol — Template symbol for generated code

Enter a template symbol name in this field. Embedded Coder software uses this symbol to add comments to the code generated from the model. For more information, see “Add Global Comments in the Generated Code” (Embedded Coder).

Dependencies

For comments to appear in the generated code, you must also set the Document type to Text.

Programmatic Use

Block Parameter: ECoderFlag
Type: character vector
Values: Abstract | Description | History | Modified History | Notes
Default: '0'

Document type — Type of document

Text (default) | RTF | HTML

Select the type of document associated with the DocBlock. The options are:

- Text
- RTF
- HTML

Dependencies

If you are using a DocBlock to add comments to your code during code generation, ensure that you set the Document Type as Text. If you set the Document Type as RTF or HTML, your comments will not appear in the code.

Programmatic Use

Block Parameter: DocumentType
Type: character vector
Values: Text | RTF | HTML
Default: 'Text'
### Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>no</td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
</tr>
</tbody>
</table>

### Extended Capabilities

#### C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

Uses the template symbol you specify for the Embedded Coder Flag block parameter to add comments to generated code. Requires an Embedded Coder license. For more information, see “Use a Simulink DocBlock to Add a Comment” (Embedded Coder).

#### HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic. See also “Generate Code with Annotations or Comments” (HDL Coder) and “Integrate Custom HDL Code Using DocBlock” (HDL Coder).

### HDL Architecture

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Annotation (default)</td>
<td>Insert text as comment in the generated code.</td>
</tr>
</tbody>
</table>
### Architecture

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HDLText</td>
<td>Integrate text as custom HDL code.</td>
</tr>
<tr>
<td>No HDL</td>
<td>Do not generate HDL code for this block.</td>
</tr>
</tbody>
</table>

### HDL Block Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConstrainedOutputPipeline</td>
<td>Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. For more details, see “ConstrainedOutputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td>InputPipeline</td>
<td>Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “InputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td>OutputPipeline</td>
<td>Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “OutputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td>TargetLanguage</td>
<td>Language of the text, either Verilog® or VHDL®. The default is VHDL.</td>
</tr>
</tbody>
</table>

When Architecture is HDLText, this property is available. To learn more, see “Integrate Custom HDL Code Using DocBlock” (HDL Coder).

### Restrictions

- **Document type** must be Text.

  HDL Coder does not support the HTML or RTF options.

- You can have a maximum of two DocBlock blocks with Architecture set to HDLText in the same subsystem.

If you have two DocBlock blocks, one must have TargetLanguage set to VHDL, and the other must have TargetLanguage set to Verilog. When generating code, HDL Coder only integrates the custom code from the DocBlock that matches the target language for code generation.
PLC Code Generation
Generate Structured Text code using Simulink® PLC Coder™.

See Also
Model Info | docblock

Topics
“Add Global Comments in the Generated Code” (Embedded Coder)

Introduced before R2006a
Dot Product

Generate dot product of two vectors

**Library:**
Simulink / Math Operations
HDL Coder / Math Operations

**Description**

The Dot Product block generates the dot product of the input vectors. The scalar output, $y$, is equal to the MATLAB operation

$$y = \text{sum}(\text{conj}(u_1) \cdot u_2)$$

where $u_1$ and $u_2$ represent the input vectors. The inputs can be vectors, column vectors (single-column matrices), or scalars. If both inputs are vectors or column vectors, they must be the same length. If $u_1$ and $u_2$ are both column vectors, the block outputs the equivalent of the MATLAB expression $u_1' * u_2$.

The elements of the input vectors can be real- or complex-valued signals. The signal type (complex or real) of the output depends on the signal types of the inputs.

<table>
<thead>
<tr>
<th>Input 1</th>
<th>Input 2</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>real</td>
<td>real</td>
<td>real</td>
</tr>
<tr>
<td>real</td>
<td>complex</td>
<td>complex</td>
</tr>
<tr>
<td>complex</td>
<td>real</td>
<td>complex</td>
</tr>
<tr>
<td>complex</td>
<td>complex</td>
<td>complex</td>
</tr>
</tbody>
</table>
Ports

Input

**Port_1 — First operand input signal**
scalar | vector

Signal representing the first operand to the dot product calculation.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

**Port_2 — Second operand input signal**
scalar | vector

Signal representing the second operand to the dot product calculation.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

Output

**Port_1 — Dot product output signal**
scalar | vector

Output signal resulting from the dot product calculation of the two input signals.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

Parameters

**Require all inputs to have the same data type — Require all inputs to have the same data type**
on (default) | off

Clear this check box for all the inputs to have different data types.

Programmatic Use

**Block Parameter:** InputSameDT
Type: character vector
Values: 'on' | 'off'
Default: 'on'

Output minimum — Minimum output value for range checking
[] (default) | scalar

Lower value of the output range that Simulink checks.

Simulink uses the minimum to perform:

- Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters”) for some blocks.
- Simulation range checking (see “Signal Ranges” and “Enable Simulation Range Checking”).
- Automatic scaling of fixed-point data types.
- Optimization of the code that you generate from the model. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. For more information, see “Optimize using the specified minimum and maximum values” (Simulink Coder).

**Note** Output minimum does not saturate or clip the actual output signal. Use the Saturation block instead.

Programmatic Use
Block Parameter: OutMin
Type: character vector
Values: '[]' | scalar
Default: '[]'

Output maximum — Maximum output value for range checking
[] (default) | scalar

Specify the upper value of the output range that Simulink checks as a finite, real, double, scalar value.

**Note** If you specify a bus object as the data type for this block, do not set the maximum value for bus data on the block. Simulink ignores this setting. Instead, set the maximum
values for bus elements of the bus object specified as the data type. For information on the Maximum parameter for a bus element, see Simulink.BusElement.

Simulink uses the maximum value to perform:

- Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters”) for some blocks.
- Simulation range checking (see “Signal Ranges” and “Enable Simulation Range Checking”).
- Automatic scaling of fixed-point data types.
- Optimization of the code that you generate from the model. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. For more information, see “Optimize using the specified minimum and maximum values” (Simulink Coder).

**Note** Output maximum does not saturate or clip the actual output signal. Use the Saturation block instead.

**Programmatic Use**

**Block Parameter:** OutMax  
**Type:** character vector  
**Values:** scalar  
**Default:** '[]'

**Output data type — Specify the output data type**

Inherit: Inherit via internal rule (default)|Inherit: Inherit via back propagation|Inherit: Same as first input|double|single|int8|uint8|int16|uint16|int32|uint32|int64|uint64|fixdt(1,16)|fixdt(1,16,0)|fixdt(1,16,2^0,0)|<data type expression>

Choose the data type for the output. The type can be inherited, specified directly, or expressed as a data type object such as Simulink.NumericType. For more information, see “Control Signal Data Types”.

When you select an inherited option, the block behaves as follows:

- **Inherit: Inherit via internal rule** — Simulink chooses a data type to balance numerical accuracy, performance, and generated code size, while taking into
account the properties of the embedded target hardware. If you change the embedded
target settings, the data type selected by the internal rule might change. For example,
if the block multiplies an input of type int8 by a gain of int16 and ASIC/FPGA is
specified as the targeted hardware type, the output data type is sfix24. If
Unspecified (assume 32-bit Generic), in other words, a generic 32-bit
microprocessor, is specified as the target hardware, the output data type is int32. If
none of the word lengths provided by the target microprocessor can accommodate the
output range, Simulink software displays an error in the Diagnostic Viewer.

It is not always possible for the software to optimize code efficiency and numerical
accuracy at the same time. If the internal rule doesn’t meet your specific needs for
numerical accuracy or performance, use one of the following options:

- Specify the output data type explicitly.
- Use the simple choice of Inherit: Same as input.
- Explicitly specify a default data type such as fixdt(1,32,16) and then use the
  Fixed-Point Tool to propose data types for your model. For more information, see
  fxptdlg.
- To specify your own inheritance rule, use Inherit: Inherit via back
  propagation and then use a Data Type Propagation block. Examples of how to
  use this block are available in the Signal Attributes library Data Type Propagation
  Examples block.
- Inherit: Inherit via back propagation — Use data type of the driving block.
- Inherit: Same as first input — Use data type of first input signal.

**Programmatic Use**

**Block Parameter:** OutDataTypeStr

**Type:** character vector

**Values:**
- 'Inherit: Inherit via internal rule'
- 'Inherit: Same as first input'
- 'Inherit: Inherit via back propagation'
- 'double'
- 'single'
- 'int8'
- 'uint8'
- 'int16'
- 'uint16'
- 'int32'
- 'uint32'
- 'int64'
- 'uint64'
- 'fixdt(1,16)'
- 'fixdt(1,16,0)'
- 'fixdt(1,16,2^0,0)'
- '<data type expression>'

**Default:** 'Inherit: Inherit via internal rule'

**Lock output data type setting against changes by the fixed-point tools — Prevent fixed-point tools from overriding data types**

- off (default)
- on
Select to lock the output data type setting of this block against changes by the Fixed-Point Tool and the Fixed-Point Advisor. For more information, see “Use Lock Output Data Type Setting” (Fixed-Point Designer).

**Programmatic Use**

**Block Parameter**: LockScale  
**Type**: character vector  
**Values**: 'off' | 'on'  
**Default**: 'off'

**Integer rounding mode — Rounding mode for fixed-point operations**

Floor (default) | Ceiling | Convergent | Nearest | Round | Simplest | Zero

Specify the rounding mode for fixed-point operations. For more information, see “Rounding” (Fixed-Point Designer).

Block parameters always round to the nearest representable value. To control the rounding of a block parameter, enter an expression using a MATLAB rounding function into the mask field.

**Programmatic Use**

**Block Parameter**: RndMeth  
**Type**: character vector  
**Values**: 'Ceiling' | 'Convergent' | 'Floor' | 'Nearest' | 'Round' | 'Simplest' | 'Zero'  
**Default**: 'Floor'

**Saturate on integer overflow — Method of overflow action**

off (default) | on

Specify whether overflows saturate or wrap.

- **off** — Overflows wrap to the appropriate value that the data type can represent.

  For example, the number 130 does not fit in a signed 8-bit integer and wraps to -126.

- **on** — Overflows saturate to either the minimum or maximum value that the data type can represent.

  For example, an overflow associated with a signed 8-bit integer can saturate to -128 or 127.

**Tip**
Consider selecting this check box when your model has a possible overflow and you want explicit saturation protection in the generated code.

Consider clearing this check box when you want to optimize efficiency of your generated code.

Clearing this check box also helps you to avoid overspecifying how a block handles out-of-range signals. For more information, see “Check for Signal Range Errors”.

When you select this check box, saturation applies to every internal operation on the block, not just the output or result.

In general, the code generation process can detect when overflow is not possible. In this case, the code generator does not produce saturation code.

**Programmatic Use**

**Block Parameter:** SaturateOnIntegerOverflow  
**Type:** character vector  
**Values:** 'off' | 'on'  
**Default:** 'off'

**Block Characteristics**

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>double</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear (default)</td>
<td>Generates a linear chain of adders to compute the sum of products.</td>
</tr>
<tr>
<td>Tree</td>
<td>Generates a tree structure of adders to compute the sum of products.</td>
</tr>
</tbody>
</table>

HDL Block Properties

<table>
<thead>
<tr>
<th>ConstrainedOutputPipeline</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. For more details, see “ConstrainedOutputPipeline” (HDL Coder).</td>
</tr>
</tbody>
</table>

| InputPipeline | Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “InputPipeline” (HDL Coder). |

| OutputPipeline | Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “OutputPipeline” (HDL Coder). |
**PLC Code Generation**
Generate Structured Text code using Simulink® PLC Coder™.

**Fixed-Point Conversion**
Design and simulate fixed-point systems using Fixed-Point Designer™.

**See Also**
Product | Product of Elements

*Introduced before R2006a*
Edit

Enter new value for parameter

Library: Simulink / Dashboard

Description

The Edit block allows you to type in new values for block parameters during simulation. Use the Edit block with other Dashboard blocks to build an interactive dashboard of controls and indicators for your model.

Double-clicking the Edit block does not open its dialog box during simulation and when the block is selected. To edit the block parameters, you can use the Property Inspector, or you can right-click the block and select Block Parameters from the context menu.

Connecting Dashboard Blocks

Dashboard blocks do not use ports to make connections. To connect Dashboard blocks to variables and block parameters in your model, use connect mode. Connect mode facilitates the process of connecting Dashboard blocks in your model, especially when you want to connect multiple blocks at once. If you only want to connect a single Dashboard block, you can also use the Connection table in the block dialog box to make the connection.

Tip To facilitate understanding and debugging your model, you can modify Dashboard block connections in your model during normal and accelerator mode simulations.

Note Dashboard blocks cannot connect to variables until you update your model diagram. To connect Dashboard blocks to variables or modify variable values between opening your model and running a simulation, update your model diagram using Ctrl+D.

To enter connect mode, click the Connect button that appears above your unconnected Dashboard block when you pause on it.
In connect mode, when you select one or more signals or blocks, a list of parameters and signals available for connection appears. Select a signal or parameter from the list to connect the selected Dashboard block. To connect another Dashboard block, pause on the block and click the **Connect** button above it. Then, make a selection of signals and blocks in your model, and choose a signal or parameter to connect.
When you finish connecting the Dashboard blocks in your model, click the **Exit** button in the upper-right of the canvas to exit connect mode.

**Tip** You can hide the message shown on unconnected blocks using the `set_param` function with the `ShowInitialText` block parameter.

### Parameter Logging

Tunable parameters connected to Dashboard blocks are logged to the Simulation Data Inspector, where you can view parameter values along with logged signal data. You can access logged parameter data in the MATLAB workspace by exporting the parameter data from the Simulation Data Inspector UI or by using the `Simulink.sdi.exportRun` function. For more information about exporting data with the Simulation Data Inspector UI, see “Export Data from the Simulation Data Inspector”. The parameter data is stored in a `Simulink.SimulationData.Parameter` object, accessible as an element in the exported `Simulink.SimulationData.Dataset`. 
**Limitations**

- Dashboard blocks can only connect to real scalar signals.
- You cannot use the **Connection** table to connect a Dashboard block to a block that is commented out. When you connect a Dashboard block to a commented block using connect mode, the Dashboard block does not display the connected value until you uncomment the block.
- Dashboard blocks cannot connect to signals inside referenced models.
- When you simulate a model hierarchy, Dashboard blocks inside referenced models do not update.
- Dashboard blocks do not support rapid accelerator simulation.
- Some signals are not available during accelerator mode simulations. To view such signals on a Dashboard block, mark the signal for logging.
- Parameters specified by indexing a variable array do not appear in the **Connection** table. For example, a block parameter defined as `engine(1)` using the variable `engine` does not appear in the table.

To access the parameter in the **Connection** table, assign the indexed value to a scalar variable, such as `engine_1`. Then, use the scalar variable to define the block parameter.

- When you connect a Dashboard block to a variable or parameter during simulation, the data for that variable or parameter does not get logged to the Simulation Data Inspector. To log variable and parameter data to the Simulation Data Inspector, connect the Dashboard block to the variable or parameter prior to simulation.
- When you simulate a model in external mode with the **Default parameter behavior** set to **Inlined**, Dashboard blocks can appear to change parameter and variable values. However, the change does not propagate to the simulation. For example, the Gain block displays changes made to its **Gain** parameter using the Dashboard blocks, but the **Gain** value used in the simulation does not change.

**Parameters**

**Connection** — Select a variable or block parameter to connect

variable and parameter connection options

Select the variable or block parameter to control using the **Connection** table. Populate the **Connection** table by selecting one or more blocks in your model. Select the radio
button next to the variable or parameter you want to control, and click **Apply**. To facilitate understanding and debugging your model, you can connect Dashboard blocks to variables and parameters in your model during simulation.

**Note** To see workspace variables in the connection table, update the model diagram using **Ctrl+D**.

**Programmatic Use**

To programmatically connect a Dashboard block to a tunable parameter or a variable, use a `Simulink.HMI.ParamSourceInfo` object. The `Simulink.HMI.ParamSourceInfo` object contains four properties. Some of the properties apply to connecting Dashboard blocks to parameters, and some apply to connecting Dashboard blocks to variables. Not all fields have a value for a connection because a given Dashboard block connects to either a parameter or a variable.

**Block Parameter:** Binding  
**Type:** `Simulink.HMI.ParamSourceInfo`  
**Default:** `[]`

**Align — Text alignment**  
**Center** (default) | **Left** | **Right**

Alignment of the text in the Edit block.

**Programmatic Use**  
**Block Parameter:** Alignment  
**Type:** string or character vector  
**Values:** 'Center' | 'Left' | 'Right'  
**Default:** 'Center'

**Label — Block label position**  
**Hide** (default) | **Bottom** | **Top**

Position of the block label. When the block is connected to an element in the model, the label is the name of the connected element.

**Programmatic Use**  
**Block Parameter:** LabelPosition  
**Type:** character vector  
**Values:** 'Hide' | 'Bottom' | 'Top'
Default: 'Hide'

Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>double</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>no</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

Dashboard blocks are ignored for code generation.

See Also
Display | Knob | Slider

Topics
“Tune and Visualize Your Model with Dashboard Blocks”
“Decide How to Visualize Simulation Data”

Introduced in R2017b
Enable

Add enable port to subsystem or model

**Library:**
- Simulink / Ports & Subsystems
- HDL Coder / Ports & Subsystems

Description

The Enable block allows an external signal to control execution of a subsystem or a model. To enable this functionality, add the block to a Subsystem block or at the root level of a model that is referenced in a Model block.

If you use an enable port at the root-level of a model:

- For multi-rate models, set the solver to single-tasking.
- For models with a fixed-step size, at least one block in the model must run at the specified fixed-step size rate.

Ports

Output

**Enable signal** — **External enable signal for a subsystem or model**

scalar

Enable signal attached externally to the outside of an Enable Subsystem block and passed to the inside of the subsystem. An enable signal port is added to an Enable block when you select the **Show output port** parameter.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | expression
Parameters

**States when enabling — Select block states when subsystem or model is disabled**

- **held** (default)
- **reset**

When a Subsystem block or Model block is disabled, select what happens to block states for the blocks within the subsystem or model.

- **held**
  - Hold block states at their previous values.
- **reset**
  - Reset block states to their initial conditions (zero if not defined).

**Programmatic Use**

**Block parameter:** StatesWhenEnabling

**Type:** character vector

**Values:** 'held' | 'reset'

**Default:** 'held'

**Propagate sizes of variable-size signals — Select when to propagate a variable-size signal**

- **Only when enabling** (default)
- **During execution**

Select when to propagate a variable-size signal.

- **Only when enabling**
  - Propagate a variable-size signal when reenabling a Subsystem block or Model block containing an Enable port block. When you select this option, sample time must be periodic.

- **During execution**
  - Propagate variable-size signals at each time step.

**Programmatic Use**

**Block parameter:** PropagateVarSize

**Type:** character vector

**Values:** 'Only when enabling' | 'During execution'

**Default:** 'Only when enabling'
Show output port — Control display of output port for enable signal
off (default) | on

The output port passes the enable signal attached externally to the outside of an Enable Subsystem block or enabled Model block to the inside.

- off
  Remove the output port on the Enable port block.

- on
  Display an output port on the Enable port block. Selecting this option allows the subsystem or model to process the enable signal.

Programmatic Use
Block parameter: ShowOutputPort
Type: character vector
Values: 'off' | 'on'
Default: 'off'

Enable zero-crossing detection — Control zero-crossing detection
on (default) | off

Control zero-crossing detection for a model.

- on
  Detect zero crossings.

- off
  Do not detect zero crossings.

Programmatic Use
Block parameter: ZeroCross
Type: character vector
Values: 'on' | 'off'
Default: 'on'

Port dimensions — Specify dimensions for the enable signal
1 (default) | [n] | [m n]

Specify dimensions for the enable signal attached externally to a Model block and passed to the inside of the block.
Scalar signal.

\[ n \]
Vector signal of width \( n \).

\[ m \times n \]
Matrix signal having \( m \) rows and \( n \) columns.

**Programmatic Use**

**Block parameter:** PortDimensions

**Type:** character vector

**Values:** '1' | '[n]' | '[m n]'

**Default:** '1'

**Sample time — Specify time interval**

-1 (default) | \( Ts \) | \([Ts, To]\)

Specify time interval between block method execution. See “Specify Sample Time”.

-1

Sample time inherited from the model.

\( Ts \)

Scalar where \( Ts \) is the time interval.

\([Ts, To]\)

Vector where \( Ts \) is the time interval and \( To \) is the initial time offset.

**Programmatic Use**

**Block parameter:** SampleTime

**Type:** character vector

**Values:** '-1' | 'Ts' | '[Ts, To]'

**Default:** '-1'

**Minimum — Specify minimum output value for the enable signal**

[] (default) | real scalar

Specify minimum value for the enable signal attached externally to a Model block and passed to the inside of the block.

Simulink uses this value to perform:
• Simulation range checking. See “Signal Ranges”.
• Automatic scaling of fixed-point data types.
• Optimization of generated code. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. See “Optimize using the specified minimum and maximum values” (Simulink Coder).

[]
  Unspecified minimum value.
real scalar
  Real double scalar value.

**Programmatic Use**

**Block parameter:** OutMin
**Type:** character vector
**Values:** '[]' | '<real scalar>'
**Default:** '[]'

**Maximum — Specify maximum output value for the enable signal**

[] (default) | real scalar

Specify maximum value for the enable signal attached externally to a Model block and passed to the inside of the block.

Simulink uses this value to perform:

• Simulation range checking. See “Signal Ranges”.
• Automatic scaling of fixed-point data types.
• Optimization of generated code. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. See “Optimize using the specified minimum and maximum values” (Simulink Coder).

[]
  Unspecified maximum value.
real scalar
  Real double scalar value.

**Programmatic Use**

**Block parameter:** OutMax
Type: character vector
Values: '[]' | '<real scalar>'
Default: '[]'

**Data type — Specify output data type for the enable signal**

- double (default)
- single
- int8
- uint8
- int16
- uint16
- int32
- uint32
- int64
- uint64
- boolean
- fixdt(1,16)
- fixdt(1,16,0)
- fixdt(1,16,2^,0)
- <data type expression>

Specify data type for the enable signal attached externally to a Model block and passed to the inside of the block.

- double
  - Double-precision floating point.
- single
  - Single-precision floating point.
- int8
  - Signed 8-bit integer.
- uint8
  - Unsigned 8-bit integer.
- int16
  - Signed 16-bit integer.
- uint16
  - Unsigned 16-bit integer.
- int32
  - Signed 32-bit integer.
- uint32
  - Unsigned 32-bit integer.
- int64
  - Signed 64-bit integer.
- uint64
  - Unsigned 64-bit integer.
- boolean
  - Boolean with a value of true or false.
fixdt(1,16)
   Signed 16-bit fixed point number with binary point undefined.
fixdt(1,16,0)
   Signed 16-bit fixed point number with binary point set to zero.
fixdt(1,16,2^0,0)
   Signed 16-bit fixed point number with slope set to $2^0$ and bias set to 0.
<data type expression>
   Data type object, for example Simulink.NumericType. Do not specify a bus object as the expression.

Programmatic Use

b
Block parameter: OutDataTypeStr
Type: character vector
Values: 'double' | 'single' | 'int8' | 'uint8' | 'int16' | 'uint16' | 'int32' | 'uint32' | 'int64' | 'uint64' | 'boolean' | '<fixdt(1,16)' | 'fixdt(1,16,0)' | 'fixdt(1,16,2^0,0)' | '<data type expression>'
Default: 'double'

Mode — Select data type category
Build in (default) | Fixed point | Expression
Select data type category and display drop-down lists to help you define the data type.
Build in
   Display drop-down lists for data type and Data type override.
Fixed point
   Display drop-down lists for Signedness, Scaling, and Data type override.
Expression
   Display text box for entering an expression.

Dependency
To enable this parameter, select the Show data type assistant button.
Programmatic Use
No equivalent command-line parameter.
Interpolate data — Specify value of missing workspace data
on (default) | off

Specify value of missing workspace data when loading data from the workspace.

- **on**
  - Linearly Interpolate output at time steps for which no corresponding workspace data exists.

- **off**
  - Do not interpolate output at time steps. The current output equals the output at the most recent time step for which data exists.

**Programmatic Use**
Block parameter: Interpolate
Type: character vector
Values: 'on' | 'off'
Default: 'on'

**Block Characteristics**

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>double</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture
This block has a single, default HDL architecture.

HDL Block Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConstrainedOutputPipeline</td>
<td>Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. For more details, see “ConstrainedOutputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td>InputPipeline</td>
<td>Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “InputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td>OutputPipeline</td>
<td>Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “OutputPipeline” (HDL Coder).</td>
</tr>
</tbody>
</table>

PLC Code Generation
Generate Structured Text code using Simulink® PLC Coder™.
**Fixed-Point Conversion**
Design and simulate fixed-point systems using Fixed-Point Designer™.

**See Also**
Enabled Subsystem | Enabled and Triggered Subsystem | Subsystem

**Topics**
“Conditionally Executed Subsystems Overview”
“Using Enabled Subsystems”
“Using Enabled and Triggered Subsystems”

**Introduced before R2006a**
Enabled and Triggered Subsystem

Subsystem whose execution is enabled and triggered by external inputs

**Library:** Simulink / Ports & Subsystems

**Description**

The Enabled and Triggered Subsystem block is a Subsystem block preconfigured as a starting point for creating a subsystem that executes when both of these conditions occur:

- Enable control signal has a positive value.
- Trigger control signal has a trigger value.

Use Enabled and Triggered Subsystem blocks to model:

- Optional functionality.
- Alternative functionality.
Ports

Input

In — Signal input to a subsystem block
scalar | vector | matrix

Placing an Inport block in a subsystem block adds an external input port to the block. The
port label matches the name of the Inport block.

Use Inport blocks to get signals from the local environment.
Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 |
Boolean | fixed point | enumerated | bus

Enable — Control signal input to a subsystem block
scalar

Placing an Enable block in a subsystem block adds an external input port to the block.
Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 |
fixed point

Trigger — Control signal input to a subsystem block
scalar

Placing a Trigger block in a subsystem block adds an external input port to the block.
Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 |
fixed point

Output

Out — Signal output from a subsystem
scalar | vector | matrix

Placing an Outport block in a subsystem block adds an output port from the block. The
port label on the subsystem block is the name of the Outport block.

Use Outport blocks to send signals to the local environment.
Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 |
Boolean | fixed point | enumerated | bus
Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>bus</th>
<th>double</th>
<th>enumerated</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
<th>string</th>
</tr>
</thead>
</table>

Direct Feedthrough | no |

Multidimensional Signals | limited |

Variable-Size Signals | limited |

Zero-Crossing Detection | no |

a. Actual data type or capability support depends on block implementation.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

Actual code generation support depends on block implementation.

Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

Actual data type support depends on block implementation.

See Also

Blocks
Enable | Enabled Subsystem | Function-Call Subsystem | Subsystem | Trigger | Triggered Subsystem

1-737
Topics
“Conditionally Executed Subsystems Overview”
“Using Enabled Subsystems”
“Using Triggered Subsystems”
“Using Enabled and Triggered Subsystems”
“Using Function-Call Subsystems”

Introduced before R2006a
**Enabled Subsystem**

Subsystem whose execution is enabled by external input

**Library:**
- Simulink / Ports & Subsystems
- HDL Coder / Ports & Subsystems

**Description**

The Enabled Subsystem block is a Subsystem block preconfigured as a starting point for creating a subsystem that executes when a control signal has a positive value.

Use Enable Subsystem blocks to model:

- Discontinuities
- Optional functionality
- Alternative functionality
Ports

Input

**In — Signal input to a subsystem block**
scalar | vector | matrix

Placing an Inport block in a subsystem block adds an external input port to the block. The port label matches the name of the Inport block.

Use Inport blocks to get signals from the local environment.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus

**Enable — Control signal input to a subsystem block**
scalar | vector | matrix

Placing an Enable block in a subsystem block adds an external input port to the block and changes the block to an Enable Subsystem block.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

Output

**Out — Signal output from a subsystem**
scalar | vector | matrix

Placing an Outport block in a subsystem block adds an output port from the block. The port label on the subsystem block is the name of the Outport block.

Use Outport blocks to send signals to the local environment.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus
Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>bus</th>
<th>double</th>
<th>enumerated</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
<th>string</th>
</tr>
</thead>
</table>

| Direct Feedthrough          | no      |
| Multidimensional Signals    | limited |
| Variable-Size Signals       | limited |
| Zero-Crossing Detection     | no      |

a. Actual data type or capability support depends on block implementation.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

Actual code generation support depends on block implementation.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

Best Practices
When using enabled subsystems in models targeted for HDL code generation, it is good practice to consider the following:

- For synthesis results to match Simulink results, the Enable port must be driven by registered logic (with a synchronous clock) on the FPGA.
• Put unit delays on Enabled Subsystem output signals. Doing so prevents the code generator from inserting extra bypass registers in the HDL code.
• Enabled subsystems can affect synthesis results in the following ways:
  • In some cases, the system clock speed can drop by a small percentage.
  • Generated code uses more resources, scaling with the number of enabled subsystem instances and the number of output ports per subsystem.

**HDL Architecture**

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Module (default)</td>
<td>Generate code for the subsystem and the blocks within the subsystem.</td>
</tr>
<tr>
<td>BlackBox</td>
<td>Generate a black box interface. The generated HDL code includes only the input/output port definitions for the subsystem. Therefore, you can use a subsystem in your model to generate an interface to existing, manually written HDL code. The black-box interface generation for subsystems is similar to the Model block interface generation without the clock signals.</td>
</tr>
<tr>
<td>No HDL</td>
<td>Remove the subsystem from the generated code. You can use the subsystem in simulation, however, treat it as a “no-op” in the HDL code.</td>
</tr>
</tbody>
</table>

**HDL Block Properties**

**General**

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AdaptivePipelining</td>
<td>Automatic pipeline insertion based on the synthesis tool, target frequency, and multiplier word-lengths. The default is inherit. See also “AdaptivePipelining” (HDL Coder).</td>
</tr>
<tr>
<td>BalanceDelays</td>
<td>Detects introduction of new delays along one path and inserts matching delays on the other paths. The default is inherit. See also “BalanceDelays” (HDL Coder).</td>
</tr>
<tr>
<td>ClockRatePipelining</td>
<td>Insert pipeline registers at a faster clock rate instead of the slower data rate. The default is inherit. See also “ClockRatePipelining” (HDL Coder).</td>
</tr>
</tbody>
</table>
### General

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
<th>Default</th>
<th>Additional References</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ConstrainedOutputPipeline</strong></td>
<td>Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. For more details, see “ConstrainedOutputPipeline” (HDL Coder).</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>DistributedPipelining</strong></td>
<td>Pipeline register distribution, or register retiming. The default is off. See also “DistributedPipelining” (HDL Coder).</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>DSPStyle</strong></td>
<td>Synthesis attributes for multiplier mapping. The default is none. See also “DSPStyle” (HDL Coder).</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>FlattenHierarchy</strong></td>
<td>Remove subsystem hierarchy from generated HDL code. The default is inherit. See also “FlattenHierarchy” (HDL Coder).</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>InputPipeline</strong></td>
<td>Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “InputPipeline” (HDL Coder).</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>OutputPipeline</strong></td>
<td>Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “OutputPipeline” (HDL Coder).</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>SharingFactor</strong></td>
<td>Number of functionally equivalent resources to map to a single shared resource. The default is 0. See also “Resource Sharing” (HDL Coder).</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>StreamingFactor</strong></td>
<td>Number of parallel data paths, or vectors, that are time multiplexed to transform into serial, scalar data paths. The default is 0, which implements fully parallel data paths. See also “Streaming” (HDL Coder).</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Target Specification

This block cannot be the DUT, so the block property settings in the **Target Specification** tab are ignored.

### Restrictions

HDL Coder supports HDL code generation for enabled subsystems that meet the following conditions:
• The enabled subsystem is not the DUT.
• The subsystem is not both triggered and enabled.
• The enable signal is a scalar.
• The data type of the enable signal is either boolean or ufix1.
• Outputs of the enabled subsystem have an initial value of 0.
• All inputs and outputs of the enabled subsystem (including the enable signal) run at the same rate.
• The Show output port parameter of the Enable block is set to Off.
• The States when enabling parameter of the Enable block is set to held (i.e., the Enable block does not reset states when enabled).
• The Output when disabled parameter for the enabled subsystem output ports is set to held (i.e., the enabled subsystem does not reset output values when disabled).
• If the DUT contains the following blocks, RAMArchitecture is set to WithClockEnable:
  • Dual Port RAM
  • Simple Dual Port RAM
  • Single Port RAM
• The enabled subsystem does not contain the following blocks:
  • CIC Decimation
  • CIC Interpolation
  • FIR Decimation
  • FIR Interpolation
  • Downsample
  • Upsample
  • HDL FIFO
  • HDL Cosimulation blocks for HDL Verifier™
  • Rate Transition

**Example**

The Automatic Gain Controller example shows how you can use enabled subsystems in HDL code generation. To open the example, enter:
hdlcoder_agc

PLC Code Generation
Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

Actual data type support depends on block implementation.

See Also

Blocks
Enable | Enabled and Triggered Subsystem | Function-Call Subsystem | Subsystem | Triggered Subsystem

Topics
“Conditionally Executed Subsystems Overview”
“Using Enabled Subsystems”
“Using Triggered Subsystems”
“Using Enabled and Triggered Subsystems”
“Using Function-Call Subsystems”

Introduced before R2006a
Enumerated Constant

Generate enumerated constant value

Library: Simulink / Sources
HDL Coder / Sources

Description

The Enumerated Constant block outputs a scalar, array, or matrix of enumerated values. You can also use the Constant block to output enumerated values, but it provides block parameters that do not apply to enumerated types, such as Output minimum and Output maximum. When you need a block that outputs only constant enumerated values, use Enumerated Constant rather than Constant. For more information, see “Simulink Enumerations”.

Ports

Output

Port_1 — Enumerated constant
scalar | vector | matrix

Enumerated constant value, specified as a scalar, vector, or matrix.

Data Types: enumerated

Parameters

Output data type — Output data type
Enum: SlDemoSign (default) | Enum:<ClassName>

Specify the enumerated type from which you want the block to output one or more values. The initial value, Enum:SlDemoSign, is a dummy enumerated type that prevents a newly
cloned block from causing an error. To specify the desired enumerated type, select it from the drop-down list or enter Enum:<ClassName> in the **Output data type** field, where *ClassName* is the name of the MATLAB class that defines the type.

**Programmatic Use**

**Block Parameter:** OutDataTypeStr  
**Type:** character vector  
**Values:** 'Enum:<ClassName>'  
**Default:** 'Enum: SlDemoSign'

**Mode — Category of data to specify**

Enumerated (default)

Select the category of data to specify.

Enumerated

Enumerated data types. Selecting Enumerated enables a second menu/text box to the right, where you can enter the class name.

**Value — Enumerated value**

*SlDemoSign.*Positive (default) | Enum:<ClassName.Value>*

Specify the value or values that the block outputs. The output of the block has the same dimensions and elements as the **Value** parameter. The initial value, *SlDemoSign.*Positive, is a dummy enumerated value that prevents a newly cloned block from causing an error.

To specify the desired enumerated values, select from the drop-down list or enter any MATLAB expression that evaluates to the desired result, including an expression that uses tunable parameters. All specified values must be of the type indicated by the **Output data type**. To specify an array that includes every value in the enumerated type, use the enumeration function.

**Programmatic Use**

**Block Parameter:** Value  
**Type:** character vector  
**Values:** 'Enum:<ClassName.Value>'  
**Default:** 'SlDemoSign.*Positive'

**Sample time — Sample time**

inf (default) | scalar | vector
Specify the interval between times that the block output can change during simulation (for example, due to tuning the `Value` parameter). The default value of `inf` indicates that the block output can never change. A sample time of `inf` speeds the simulation and generated code by avoiding the need to recompute the block output. For more information, see “Specify Sample Time”.

**Programmatic Use**
**Block Parameter:** `SampleTime`  
**Type:** character vector  
**Values:** scalar | vector  
**Default:** 'inf'

### Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>enumerated</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>yes</td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
</tr>
</tbody>
</table>

### Extended Capabilities

**C/C++ Code Generation**  
Generate C and C++ code using Simulink® Coder™.

**HDL Code Generation**  
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.
HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

**HDL Architecture**

This block has a single, default HDL architecture.

**HDL Block Properties**

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConstrainedOutputPipeline</td>
<td>Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. For more details, see “ConstrainedOutputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td>OutputPipeline</td>
<td>Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “OutputPipeline” (HDL Coder).</td>
</tr>
</tbody>
</table>

**PLC Code Generation**

Generate Structured Text code using Simulink® PLC Coder™.

**See Also**

Constant | enumeration

**Topics**

“Use Enumerated Data in Simulink Models”
“Simulink Enumerations”
“Code Generation for Enumerations”
“Specify Sample Time”

**Introduced in R2009b**
Environment Controller

Create branches of block diagram that apply only to simulation or only to code generation

Library: Simulink / Signal Routing

Description

The Environment Controller block outputs the signal at its Sim port only if the model that contains it is being simulated. It outputs the signal at its Coder port only if code is being generated from the model. This option enables you to create branches of a block diagram that apply only to simulation or code generation. This table describes various scenarios where either the Sim or Coder port applies.

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal mode simulation</td>
<td>Sim</td>
</tr>
<tr>
<td>Accelerator mode simulation</td>
<td>Sim</td>
</tr>
<tr>
<td>Rapid accelerator mode simulation</td>
<td>Sim</td>
</tr>
<tr>
<td>Simulation of a referenced model in normal or accelerator modes</td>
<td>Sim</td>
</tr>
<tr>
<td>Simulation of a referenced model in processor-in-the-loop (PIL) mode</td>
<td>Coder</td>
</tr>
<tr>
<td></td>
<td>(uses the same code generated for a referenced model)</td>
</tr>
<tr>
<td>External mode simulation</td>
<td>Coder</td>
</tr>
<tr>
<td>Standard code generation</td>
<td>Coder</td>
</tr>
<tr>
<td>Code generation of a referenced model</td>
<td>Coder</td>
</tr>
</tbody>
</table>

Simulink Coder software does not generate code for blocks connected to the Sim port if these conditions hold:

- On the Code Generation > Optimization pane of the Configuration Parameters dialog box, you set Default parameter behavior to Inlined.
• The blocks connected to the Sim port do not have external signals.
• The Sim port input path does not contain an S-function or an Interpreted MATLAB Function block.

If you enable block reduction optimization, Simulink eliminates blocks in the branch connected to the Coder port when compiling the model for simulation. For more information, see “Block reduction”.

**Note** Simulink Coder code generation eliminates the blocks connected to the Sim branch only if the Sim branch has the same signal dimensions as the Coder branch. Regardless of whether it eliminates the Sim branch, Simulink Coder uses the sample times on the Sim branch as well as the Coder branch to determine the fundamental sample time of the generated code and might, in some cases, generate sample-time handling code that applies only to sample times specified on the Sim branch.

### Ports

#### Input

**Sim** — **Simulation input**
scalar | vector | matrix

Simulation input values, specified as a scalar, vector, or matrix. Input signal must have the same width as the input to the Coder port.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus

**Coder** — **Code generation input**
scalar | vector | matrix

Code generation input values, specified as a scalar, vector, or matrix. Input signal must have the same width as the input to the Sim port.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus
**Output**

*Out — Values from Sim or Coder input port*

scalar | vector | matrix

Values from the Sim or Coder input port, depending on the current environment. For more information on what the block outputs in various simulation and code generation modes, see “Description” on page 1-750.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus

**Block Characteristics**

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>bus</th>
<th>double</th>
<th>enumerated</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Extended Capabilities**

**C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

Not recommended for production code. Relates to resource limits and restrictions on speed and memory often found in embedded systems. Generated code can contain dynamic allocation and freeing of memory, recursion, additional memory overhead, and
widely-varying execution times. While the code is functionally valid and generally acceptable in resource-rich environments, smaller embedded targets often cannot support such code. Usually, blocks evolve toward being suitable for production code. Thus, blocks suitable for production code remain suitable.

**Fixed-Point Conversion**
Design and simulate fixed-point systems using Fixed-Point Designer™.

**See Also**
“Default parameter behavior” (Simulink Coder)

**Topics**
“Spawn and Synchronize Execution of RTOS Task” (Simulink Coder)
“Dual-Model Approach: Code Generation” (Simulink Coder)
“Block reduction”

**Introduced before R2006a**
Event Listener

Add event port to a subsystem block

Library: Ports & Subsystems

Description

Add event port to a Subsystem block.

Parameters

Event type — Select event type for subsystem

Initialize (default) | Terminate | Reset

Select event type for subsystem to execute initialize, reset, or terminate algorithms.

Initialize

Select to trigger the execution of an Initialize Function block with an initialize event.

Terminate

Select to trigger the execution of a Terminate Function block with a terminate event.

Reset

Select to trigger the execution of an Initialize Function block reconfigured as a Reset Function block with a reset event.

Programmatic Use

Block Parameter: EventType

Type: character vector

Value: 'Initialize' | 'Terminate' | 'Reset'

Default: 'Initialize'

Event name — Specify event name

Reset (default) | event name

Specify event name for Reset Function block
Reset

Default name on the face of the Reset Function block.

event name

User entered name displayed on the face of the Reset Function block, and the name of the reset event port on the Model block containing the Reset Function block.

When tying the name for a reset function, the auto-completion list provides some suggestions. The list is not complete.

Dependency

To enable this parameter, set the Event parameter to Reset.

Programmatic Use

**Block Parameter:** EventName  
**Type:** character vector  
**Value:** 'reset' | '<event name>'  
**Default:** 'reset'

Enable variant condition — Control activating the variant control (condition)

off (default) | on

Control activating the variant control (condition) defined with the Variant Control parameter.

☐ off

Deactivate variant control of subsystem.

☑ on

Activate variant control of subsystem.

Dependency

Selecting this parameter, enables the Variant control and Generate preprocessor conditionals parameters.

When you select Enable variant condition parameter, the badge indicates the change.
Programmatic Use

Block Parameter: Variant
Type: character vector
Value: 'off' | 'on'
Default: 'off'

Note

- The Reset event port of a Model block can be conditional and propagates the net Variant conditions defined on the corresponding Reset Function block(s) in the referenced model.
- Initialize and Terminate event ports are always unconditional because they control both the model default and block-specific initialize and terminate events of the referenced model. If you define an Initialize function block in the referenced model, it corresponds to an explicit initialize event.

Variant control — Specify variant control (condition) expression

Variant (default) | logical expression

Specify variant control (condition) expression that executes a variant Initialize function, Reset function, or Terminate function block when the expression evaluates to true.

Variant

Default name for a logical (Boolean) expression.

logical expression

A logical (Boolean) expression or a Simulink.Variant object representing a boolean expression.

If you want to generate code for your model, define the variables in the expression as Simulink.Parameter objects.

Dependency

To enable this parameter, select the Enable variant condition parameter.

Programmatic Use

Block Parameter: VariantControl
Type: character vector
**Value:** 'Variant' | '<logical expression>'

**Default:** 'Variant'

**Generate preprocessor conditionals — Select if variant choices are enclosed within C preprocessor conditional statements**

- off (default) | on

Select if variant choices are enclosed within C preprocessor conditional statements.

- **off**
  - Does not enclose variant choices within C preprocessor conditional statements.

- **on**
  - When generating code for an ERT target, encloses variant choices within C preprocessor conditional statements (#if).

**Dependency**

To enable this parameter, select the **Enable variant condition** parameter.

When you select **Generate preprocessor conditionals** parameter, the badge indicates the change.

**Programmatic Use**

**Block Parameter:** GeneratePreprocessorConditionals

- **Type:** character vector
- **Value:** 'off' | 'on'
- **Default:** 'off'

**See Also**

Initialize Function | Reset Function | State Reader | State Writer | Terminate Function
Topics
“Using Initialize, Reset, and Terminate Functions”
“Create Test Harness to Generate Function Calls”
Extract Bits

Output selection of contiguous bits from input signal

Library: Simulink / Logic and Bit Operations
        HDL Coder / Logic and Bit Operations

Description

The Extract Bits block allows you to output a contiguous selection of bits from the stored integer value of the input signal. Use the Bits to extract parameter to define the method for selecting the output bits.

- Select **Upper half** to output the half of the input bits that contain the most significant bit. If there is an odd number of bits in the input signal, the number of output bits is given by the equation

  \[
  \text{number of output bits} = \text{ceil}(\text{number of input bits}/2)
  \]

- Select **Lower half** to output the half of the input bits that contain the least significant bit. If there is an odd number of bits in the input signal, the number of output bits is given by the equation

  \[
  \text{number of output bits} = \text{ceil}(\text{number of input bits}/2)
  \]

- Select **Range starting with most significant bit** to output a certain number of the most significant bits of the input signal. Specify the number of most significant bits to output in the Number of bits parameter.

- Select **Range ending with least significant bit** to output a certain number of the least significant bits of the input signal. Specify the number of least significant bits to output in the Number of bits parameter.

- Select **Range of bits** to indicate a series of contiguous bits of the input to output in the Bit indices parameter. You indicate the range in [start end] format, and the indices of the input bits are labeled contiguously starting at 0 for the least significant bit.
This block does not report wrap on overflow warnings during simulation. To report these warnings, see the Simulink.restoreDiagnostic reference page. The block does report errors due to wrap on overflow.

**Ports**

**Input**

**Port_1 — Input signal**

scalar | vector | matrix | N-D array

Input signal, specified as a scalar, vector, matrix, or N-D array. Floating-point inputs are passed through the block unchanged. boolean inputs are treated as uint8 signals.

**Note** Performing bit operations on a signed integer is difficult. You can avoid difficulty by converting the data type of your input signals to unsigned integer types.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point

**Output**

**Port_1 — Extracted bits**

scalar | vector | matrix | N-D array

Contiguous selection of extracted bits, specified as a scalar, vector, matrix, or N-D array. Floating-point inputs are passed through the block unchanged.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

**Parameters**

**Bits to extract — Method for extracting bits**

Upper half (default) | Lower half | Range starting with most significant bit | Range ending with least significant bit | Range of bits
Select the method for extracting bits from the input signal.

Consider an input signal that is represented in binary by 110111001:

- If you select Upper half for the Bits to extract parameter, the output is 11011 in binary.
- If you select Lower half for the Bits to extract parameter, the output is 11001 in binary.
- If you select Range starting with most significant bit for the Bits to extract parameter, and specify 3 for the Number of bits parameter, the output is 110 in binary.
- If you select Range ending with least significant bit for the Bits to extract parameter, and specify 8 for the Number of bits parameter, the output is 10111001 in binary.
- If you select Range of bits for the Bits to extract parameter, and specify [4 7] for the Bit indices parameter, the output is 1011 in binary.

Programmatic Use
Block Parameter: bitsToExtract
Type: character vector
Values: 'Upper half' | 'Lower half' | 'Range starting with most significant bit' | 'Range ending with least significant bit' | 'Range of bits'
Default: 'Upper half'

Number of bits — Number of bits to output
8 (default) | positive integer

Select the number of bits to output from the input signal. Signed integer data types must have at least two bits. Unsigned data integer types can be as short as a single bit.

Dependencies
To enable this parameter, set Bits to extract to Range starting with most significant bit or Range ending with least significant bit.

Programmatic Use
Block Parameter: numBits
Type: character vector
Values: positive integer
Default: '8'
**Bit indices — Contiguous range of bits to output**

Specify a contiguous range of bits of the input signal to output. Specify the range in [start end] format. The indices are assigned to the input bits starting with 0 at the least significant bit.

**Dependencies**

To enable this parameter, set **Bits to extract** to **Range of bits**.

**Programmatic Use**

**Block Parameter:** `bitIdxRange`  
**Type:** character vector  
**Values:** contiguous range  
**Default:** 

**Output scaling mode — Output scaling mode**

Select the scaling mode to use on the output bit selection:

- When you select **Preserve fixed-point scaling**, the fixed-point scaling of the input is used to determine the output scaling during the data type conversion.
- When you select **Treat bit field as an integer**, the fixed-point scaling of the input is ignored, and only the stored integer is used to compute the output data type.

**Programmatic Use**

**Block Parameter:** `outScalingMode`  
**Type:** character vector  
**Values:** 

**Block Characteristics**

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>double</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>--------------------------</td>
<td>-----</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

### HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

### HDL Architecture

This block has a single, default HDL architecture.

### HDL Block Properties

<table>
<thead>
<tr>
<th>ConstrainedOutputPipeline</th>
<th>Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. For more details, see “ConstrainedOutputPipeline” (HDL Coder).</th>
</tr>
</thead>
<tbody>
<tr>
<td>InputPipeline</td>
<td>Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “InputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td>OutputPipeline</td>
<td>Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “OutputPipeline” (HDL Coder).</td>
</tr>
</tbody>
</table>

**PLC Code Generation**
Generate Structured Text code using Simulink® PLC Coder™.

**Fixed-Point Conversion**
Design and simulate fixed-point systems using Fixed-Point Designer™.

**See Also**
Bit Clear | Bit Set | Bitwise Operator

Introduced before R2006a
**Fcn**

Apply specified expression to input  

**Library:** Simulink / User-Defined Functions

---

**Description**

The Fcn block applies the specified mathematical expression to its input. The expression can include one or more of these components:

- **u** — The input to the block. If $u$ is a vector, $u(i)$ represents the $i$th element of the vector; $u(1)$ or $u$ alone represents the first element.
- Numeric constants.
- Arithmetic operators (+ - * / ^).
- Relational operators (== != > < >= <=) — The expression returns 1 if the relation is true; otherwise, it returns 0.
- Logical operators (&& || !) — The expression returns 1 if the relation is true; otherwise, it returns 0.
- Parentheses.
- Mathematical functions — abs, acos, asin, atan, atan2, ceil, cos, cosh, exp, floor, hypot, log, log10, power, rem, sgn (equivalent to sign in MATLAB), sin, sinh, sqrt, tan, and tanh.

**Note** The Fcn block does not support round and fix. Use the Rounding Function block to apply these rounding modes.

- Workspace variables — Variable names that are not recognized in the preceding list of items are passed to MATLAB for evaluation. Matrix or vector elements must be specifically referenced (e.g., $A(1,1)$ instead of $A$ for the first element in the matrix).

The Fcn block observes the following rules of operator precedence:
The expression differs from a MATLAB expression in that the expression cannot perform matrix computations. Also, this block does not support the colon operator (:).

Block input can be a scalar or vector. The output is always a scalar. For vector output, consider using the Math Function block. If a block input is a vector and the function operates on input elements individually (for example, the sin function), the block operates on only the first vector element.

Limitations

• You cannot tune the expression during simulation in Normal or Accelerator mode (see “How Acceleration Modes Work”), or in generated code. To implement tunable expressions, tune the expression outside the Fcn block. For example, use the Relational Operator block to evaluate the expression outside.

• The Fcn block does not support custom storage classes. See “Apply Built-In and Customized Storage Classes to Data Elements” (Embedded Coder).

Ports

Input

In — Input to a Fcn block

scalar | vector
The Fcn block accepts and outputs signals of type single or double.

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Data Types: single | double

**Output**

**Out — Output from a Fcn block**

scalar

The Fcn block accepts and outputs signals of type single or double.

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Data Types: single | double

**Parameters**

**Expression — Specify the mathematical expression**

mathematical expression

Specify the mathematical expression to apply to the input. Expression components are listed above. The expression must be mathematically well-formed (uses matched parentheses, proper number of function arguments, and so on). The expression has restrictions on tunability (see “Limitations” on page 1-766).

**Programmatic Use**

**Block Parameter:** Expr

**Type:** character vector

**Value:** mathematical expression

**Default:** 'sin(u(1)*exp(2.3*(-u(2))))'

**Sample time — Specify sample time in the block**

scalar

**Note** This parameter is not visible in the block dialog box unless it is explicitly set to a value other than -1. To learn more, see “Blocks for Which Sample Time Is Not Recommended”.

1-767
### Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>double</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
</tr>
</tbody>
</table>

### Extended Capabilities

#### C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

#### HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

You can generate HDL code for the block with both single and double data types in native floating-point mode. See “Generate Target-Independent HDL Code with Native Floating-Point” (HDL Coder).

**HDL Architecture**

This block has a single, default HDL architecture.
## HDL Block Properties

<table>
<thead>
<tr>
<th>General</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>BalanceDelays</strong></td>
<td>Detects introduction of new delays along one path and inserts matching delays on the other paths. The default is inherit. See also “BalanceDelays” (HDL Coder).</td>
</tr>
<tr>
<td><strong>ClockRatePipelining</strong></td>
<td>Insert pipeline registers at a faster clock rate instead of the slower data rate. The default is inherit. See also “ClockRatePipelining” (HDL Coder).</td>
</tr>
<tr>
<td><strong>ConstrainedOutputPipeline</strong></td>
<td>Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. For more details, see “ConstrainedOutputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td><strong>DistributedPipelining</strong></td>
<td>Pipeline register distribution, or register retiming. The default is off. See also “DistributedPipelining” (HDL Coder).</td>
</tr>
<tr>
<td><strong>FlattenHierarchy</strong></td>
<td>Remove subsystem hierarchy from generated HDL code. The default is inherit. See also “FlattenHierarchy” (HDL Coder).</td>
</tr>
<tr>
<td><strong>InputPipeline</strong></td>
<td>Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “InputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td><strong>OutputPipeline</strong></td>
<td>Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “OutputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td><strong>SharingFactor</strong></td>
<td>Number of functionally equivalent resources to map to a single shared resource. The default is 0. See also “Resource Sharing” (HDL Coder).</td>
</tr>
</tbody>
</table>

## PLC Code Generation
Generate Structured Text code using Simulink® PLC Coder™.

## See Also
Interpreted MATLAB Function | MATLAB Function | MATLAB System
Introduced before R2006a
**Find Nonzero Elements**

Find nonzero elements in array  
**Library:** Simulink / Math Operations

**Description**

The Find Nonzero Elements block locates all nonzero elements of the input signal and returns the linear indices of those elements. If the input is a multidimensional signal, the Find Nonzero Elements block can also return the subscripts of the nonzero input elements. In both cases, you can show an output port with the nonzero input values.

The Find Nonzero Elements block outputs a variable-size signal. The sample time for any variable-size signal must be discrete. If your model does not already use a fixed-step solver, you may need to select a fixed-step solver in the Configuration Parameters dialog. For more information, see “Compare Solvers” and “Choose a Solver”.

**Ports**

**Input**

**Port_1 — Input signal**

scalar | vector | matrix | N-D array

Input signal from which the block finds all nonzero elements.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point

**Output**

**Port_1 — Indices of nonzero elements**

variable-size signal
The Find Nonzero Elements block outputs the indices of nonzero elements as a variable-size signal. You control the data type of the output using the **Output data type** block parameter.

**Dependencies**

By default, the block outputs linear indices from the first output port. When you change the **Index output format** to Subscripts, the block instead provides the element indices of a two-dimension or larger signal in a subscript form. In this mode, you must specify the **Number of input dimensions**, and the block creates a separate output port for each dimension.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point

**Port 2 — Values of nonzero elements**

variable-size signal

The Find block can optionally output the values of all nonzero elements as a variable-size signal.

**Dependencies**

To enable this port, select **Show output port for nonzero input values**.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point

**Parameters**

**Main**

**Index output format — Format for indices of nonzero elements**

Linear indices (default) | Subscripts

Select the output format for the indices of the nonzero input values.

- Selecting **Linear indices** provides the element indices of any dimension signal in a vector form. For one dimension (vector) signals, indices correspond to the position of nonzero values within the vector. For signals with more than one dimension, the conversion of subscripts to indices is along the first dimension. You do not need to know the signal dimension of the input signal.
• Selecting Subscripts provides the element indices of a two-dimension or larger signal in a subscript form. Because the block shows an output port for each dimension, this option requires you to specify the **Number of input dimensions**.

**Programmatic Use**

**Block Parameter:** IndexOutputFormat  
**Type:** character vector  
**Values:** 'Linear indices' | 'Subscripts'  
**Default:** 'Linear indices'

**Number of input dimensions — Number of dimensions for the input signal**  
1 (default) | scalar

Specify the number of dimensions for the input signal as a positive integer value from 1 to 32.

**Dependencies**

To enable this parameter, set **Index output format** to **Subscripts**.

**Programmatic Use**

**Block Parameter:** NumberOfInputDimensions  
**Type:** character vector  
**Values:** scalar  
**Default:** '1'

**Index mode — Specify zero- or one-based indexing**  
Zero-based (default) | One-based

Specify the indexing mode as **Zero-based** or **One-based**.

• For **Zero-based** indexing, an index of 0 specifies the first element of the input vector. An index of 1 specifies the second element, and so on.

• For **One-based** indexing, an index of 1 specifies the first element of the input vector. An index of 2, specifies the second element, and so on.

**Programmatic Use**

**Block Parameter:** IndexMode  
**Type:** character vector  
**Values:** 'Zero-based' | 'One-based'  
**Default:** 'Zero-based'
Show output port for nonzero input values — Enable output port for nonzero values
off (default) | on

Show or hide the output port for nonzero input values.

- When you clear this check box (off), the block hides the output port for nonzero input values.
- When you select this check box (on), the block displays the output port for nonzero input values. The additional output port provides values of the nonzero input elements.

Programmatic Use
Block Parameter: ShowOutputPortForNonzeroInputValues
Type: character vector
Values: 'on' | 'off'
Default: 'off'

Sample time — Specify sample time as a value other than -1
-1 (default) | scalar | vector

Specify the sample time as a value other than -1. For more information, see “Specify Sample Time”.

Dependencies
This parameter is not visible unless it is explicitly set to a value other than -1. To learn more, see “Blocks for Which Sample Time Is Not Recommended”.

Programmatic Use
Block Parameter: SampleTime
Type: character vector
Values: scalar or vector
Default: '-1'

Data Types
Output data type — Output data type
Inherit: Inherit via internal rule (default) | int8 | uint8 | int16 | uint16 | int32 | uint32 | int64 | uint64 | fixdt(1,16) | <data type expression>

Specify the output data type.
Click the **Show data type assistant** button to display the **Data Type Assistant**, which helps you set the data type attributes. For more information, see “Specify Data Types Using Data Type Assistant”.

**Programmatic Use**

**Block Parameter:** `OutDataTypeStr`  
**Type:** character vector  
**Values:**  
- 'Inherit: Inherit via internal rule'  
- 'int8'  
- 'uint8'  
- 'int16'  
- 'uint16'  
- 'int32'  
- 'uint32'  
- 'int64'  
- 'uint64'  
- 'fixdt(1,16)'  
- '<data type expression>'  
**Default:** 'Inherit: Inherit via internal rule'

**Block Characteristics**

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>double</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Extended Capabilities**

**C/C++ Code Generation**
Generate C and C++ code using Simulink® Coder™.

**Fixed-Point Conversion**
Design and simulate fixed-point systems using Fixed-Point Designer™.
See Also
find

Topics
“Array Indexing” (MATLAB)
“Variable-Size Signal Basics”
“Simulink Models Using Variable-Size Signals”
“Control Signal Data Types”

Introduced in R2010a
First Order Hold

Implement linearly extrapolated first order hold on input signal

Library: Simulink / Continuous

Description

The First Order Hold block outputs a continuous piecewise linear approximation of its input signal. Use the First Order Hold block to convert a sampled discrete signal to a continuous signal without triggering a solver reset.

You can also use the First Order Hold block to break algebraic loops in your model.

Ports

Input

Port 1 — Input signal
scalar | vector | matrix

Input signal, specified as a real-valued scalar, vector, or matrix.

Data Types: double

Output

Port 1 — Linearly approximated output signal
scalar | vector | matrix

Piecewise linear approximation of the input signal.

Data Types: double
Parameters

Initial output — Initial output
0 (default) | scalar | vector

Specify the output that the block generates until the simulation time exceeds the first sample hit time of the discrete input.

Dependencies

• The initial output of this block cannot be inf or NaN.
• A Run-to-run tunable parameter cannot be changed during simulation run time. However, changing it before a simulation begins does not cause Accelerator or Rapid Accelerator to regenerate code.

Programmatic Use

Block Parameter: InitialOutput
Type: character vector, string
Values: scalar | vector
Default: '0'

Output algorithm — Approximation technique for output signal
Slow (Avoid Overshoot) (default) | Fast (Minimize Error)

Specify the approximation technique to use to smooth the input signal. The slow, or low pass algorithm aims to avoid overshoots when smoothing the input signal. The fast, or high pass algorithm minimizes error by attempting to predict the next input during the update method of the block. This error reduction in predicted output of the block is traded off against a tendency to overshoot the reference, or input signal.

Programmatic Use

Block Parameter: OutputAlgorithm
Type: character vector, string
Values: Slow (Avoid Overshoot) | Fast (Minimize Error)
Default: 'Slow (Avoid Overshoot)'

Reset if relative extrapolation error exceeds — Error tolerance for solver reset
inf (default) | nonnegative real scalar
Specify the tolerance level for the extrapolation error of the Output Algorithm. Extrapolation error greater than this value will cause Simulink to reset the solver for the model.

**Tip** Enabling the *Allow continuous input* parameter when this parameter is set to \( \text{inf} \) can lead to numerical inaccuracies in simulation.

**Programmatic Use**

**Block Parameter:** ErrorTolerance  
**Type:** character vector, string  
**Values:** scalar  
**Default:** 'inf'

**Allow continuous input — Apply hold on continuous signals**  
off (default) | on

Select this parameter to enable the block to accept continuous signals as input. Enabling this parameter creates a delay between the input and output continuous signals.

**Note** Enable this parameter when using this block to break an algebraic loop.

**Programmatic Use**

**Block Parameter:** AllowContinuousInput  
**Type:** character vector, string  
**Values:** 'off' | 'on'  
**Default:** 'off'

### Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>double</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>no</td>
</tr>
</tbody>
</table>
Algorithms

Slow (Avoid overshoot)

The linear extrapolation performed by the slow or low pass algorithm can be described as:

\[ y(t) = M_k(t - T_k) + C_k \quad \forall \ T_k \leq t < t_{k+1} \]

where:

- \( T_k \) is the time at \( k^{th} \) major step.
- \( U_k \) is the input at \( k^{th} \) major step.
- \( C_k \) and \( M_k \) are the linear extrapolation coefficients.
- \( y_k \) is the continuous output at time \( T_k \).

The coefficients, or slope \( M_k \) and \( y \)-intercept \( C_k \), are calculated as follows

\[
H_k = T_k - T_{k-1} \\
M_{k+1} = \frac{U_k - U_{k-1}}{H_k} \\
C_k = y_k
\]

Fast (Minimize Error)

The linear extrapolation performed by the fast, or high pass, algorithm can be described as:

\[ y(t) = N_k(t - T_k) + C_k \quad \forall \ T_k \leq t < T_{k+1} \]

where:
• $T_k$ is the time at $k^{th}$ major step.
• $U_k$ is the input at $k^{th}$ major step.
• $\hat{U}$ is the predicted value of the input at $(k+1)^{th}$ major step.
• $(C_k$ and $N_k$ are the linear extrapolation coefficients.
• $y_k$ is the continuous output at time $T_k$.

The coefficients, or slope $N_k$ and $y$-intercept $C_k$, are calculated as follows

$$H_k = T_k - T_{k-1}$$
$$M_{k+1} = \frac{U_k - U_{k-1}}{H_k}$$
$$\hat{U} = U_k + M_{k+1}(H_K)$$
$$N_{k+1} = \frac{\hat{U} - y_k}{H_K}$$
$$C_k = y_k$$

The fast algorithm minimizes error by attempting to predict the next input during the update method of the block. This error reduction in predicted output of the block is traded off against a tendency to overshoot the input signal.

**Extended Capabilities**

**C/C++ Code Generation**
Generate C and C++ code using Simulink® Coder™.

Not recommended for production code generation.
Consider discretizing your model.

**See Also**
Memory | Zero-Order Hold
Introduced in R2019b
Fixed-Point State-Space

Implement discrete-time state space

Library:  Simulink / Additional Math & Discrete / Additional Discrete

Description

The Fixed-Point State-Space block implements the system described by

\[
y(n) = Cx(n) + Du(n) \\
x(n + 1) = Ax(n) + Bu(n)
\]

where \( u \) is the input, \( x \) is the state, and \( y \) is the output. Both equations have the same data type.

- \( A \) must be an \( n \)-by-\( n \) matrix, where \( n \) is the number of states.
- \( B \) must be an \( n \)-by-\( m \) matrix, where \( m \) is the number of inputs.
- \( C \) must be an \( r \)-by-\( n \) matrix, where \( r \) is the number of outputs.
- \( D \) must be an \( r \)-by-\( m \) matrix.

In addition:

- The state \( x \) must be an \( n \)-by-1 vector.
- The input \( u \) must be an \( m \)-by-1 vector.
- The output \( y \) must be an \( r \)-by-1 vector.

The block accepts one input and generates one output. The width of the input vector is the number of columns in the \( B \) and \( D \) matrices. The width of the output vector is the number of rows in the \( C \) and \( D \) matrices. To define the initial state vector, use the **Initial conditions** parameter.
Ports

Input

Port_1 — Input signal
scalar | vector | matrix

Input vector, where the width equals the number of columns in the B and D matrices. For more information, see “Description” on page 1-783.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 |
Boolean | fixed point

Output

Port_1 — Output signal
scalar | vector | matrix

Output signal, with width equal to the number of rows in the C and D matrices. For more information, see “Description” on page 1-783.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 |
Boolean | fixed point

Parameters

Main

State Matrix A — Matrix of states
[2.6020 -2.2793 0.6708; 1 0 0; 0 1 0] (default) | scalar | vector | matrix

Specify the matrix of states as an n-by-n matrix of real or complex values, where n is the number of states. For more information on the matrix coefficients, see “Description” on page 1-783.

Programmatic Use
Block Parameter: A
Type: character vector
**Values:** scalar | vector | matrix
**Default:** ' [2.6020  -2.2793  0.6708;  1  0  0;  0  1  0] '

**Input Matrix B — Column vector of inputs**
[ 1;  0;  0] (default) | scalar | vector | matrix

Specify the column vector of inputs as an $n$-by-$m$ matrix of real or complex values, where $n$ is the number of states, and $m$ is the number of inputs. For more information on the matrix coefficients, see “Description” on page 1-783.

**Programmatic Use**
**Block Parameter:** B
**Type:** character vector
**Values:** scalar | vector | matrix
**Default:** ' [ 1;  0;  0] '

**Output Matrix C — Column vector of outputs**
[ 0.0184  0.0024  0.0055] (default) | scalar | vector | matrix

Specify the column vector of outputs as an $r$-by-$n$ matrix of real or complex values, where $r$ is the number of outputs, and $n$ is the number of states. For more information on the matrix coefficients, see “Description” on page 1-783.

**Programmatic Use**
**Block Parameter:** C
**Type:** character vector
**Values:** scalar | vector | matrix
**Default:** ' [ 0.0184  0.0024  0.0055] '

**Direct Feedthrough Matrix D — Matrix for direct feedthrough**
[ 0.0033] (default) | scalar | vector | matrix

Specify the matrix for direct feedthrough as an $r$-by-$m$ matrix of real or complex values, where $r$ is the number of outputs, and $m$ is the number of inputs. For more information on the matrix coefficients, see “Description” on page 1-783.

**Programmatic Use**
**Block Parameter:** D
**Type:** character vector
**Values:** scalar | vector | matrix
**Default:** ' [ 0.0033] '
Initial condition for state — Initial state vector
0.0 (default) | scalar | vector | matrix

Specify the initial condition for the state.

Limitations

The initial state cannot be inf or NaN.

Programmatic Use
Block Parameter: X0
Type: character vector
Values: scalar | vector | matrix
Default: '0.0'

Signal Attributes

Data type for internal calculations — Data type for internal calculations
fixdt('double') (default) | data type string

Specify the data type the block uses for internal calculations.

Programmatic Use
Block Parameter: InternalDataType
Type: character vector
Values: data type string
Default: 'fixdt('double')'

Scaling for State Equation AX+BU — Scaling for state equations
2^0 (default) | scalar

Specify the scaling for the state equation AX+BU.

Programmatic Use
Block Parameter: StateEqScaling
Type: character vector
Values: scalar
Default: '2^0'

Scaling for Output Equation CX+DU — Scaling for output equations
2^0 (default) | scalar

Specify the scaling for the output equation CX+DU.
Programmatic Use

**Block Parameter:** InternalDataType
**Type:** character vector
**Values:** scalar
**Default:** '2^0'

**Lock output data type setting against changes by the fixed-point tools — Prevent fixed-point tools from overriding data types**

off (default) | on

Select to lock the output data type setting of this block against changes by the Fixed-Point Tool and the Fixed-Point Advisor. For more information, see “Use Lock Output Data Type Setting” (Fixed-Point Designer).

**Programmatic Use**

**Block Parameter:** LockScale
**Type:** character vector
**Values:** 'off' | 'on'
**Default:** 'off'

**Integer rounding mode — Rounding mode for fixed-point operations**

Floor (default) | Ceiling | Convergent | Nearest | Round | Simplest | Zero

Specify the rounding mode for fixed-point operations. For more information, see “Rounding” (Fixed-Point Designer).

**Programmatic Use**

**Block Parameter:** RndMeth
**Type:** character vector
**Values:** 'Ceiling' | 'Convergent' | 'Floor' | 'Nearest' | 'Round' | 'Simplest' | 'Zero'
**Default:** 'Floor'

**Saturate to max or min when overflows occur — Method of overflow action**

off (default) | on

When you select this check box, overflows saturate to the maximum or minimum value that the data type can represent. Otherwise, overflows wrap.

When you select this check box, saturation applies to every internal operation on the block, not just the output or result. In general, the code generation process can detect when overflow is not possible. In this case, the code generator does not produce saturation code.
Programmatic Use
Block Parameter: DoSatur
Type: character vector
Values: 'off' | 'on'
Default: 'off'

Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean(^a)</th>
<th>double</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\(^a\) This block is not recommended for use with Boolean signals.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

See Also
Discrete State-Space | State-Space
Topics
“State”
“Fixed-Point Numbers”

Introduced before R2006a
Floating Scope and Scope Viewer

Display signals generated during simulation without signal lines

Library:
- Simulink / Sinks
- HDL Coder / Sinks

Description

The Simulink Scope Viewer and Floating Scope block display time domain signals with respect to simulation time. The Scope Viewer and Floating Scope block have the same functionality as the Scope block, but they are not connected to signal lines.
For information on controlling a Floating Scope block from the command line, see “Control Scope Blocks Programmatically” in the Simulink documentation.

**Connect Signals**

In the Floating Scope window or Scope viewer window, use the button to connect signals.

To add a viewer with a connected signal, select a signal in the model. Then, on the Simulation tab, in the Prepare gallery, click Add Viewer.

For more details, see “Add Signals to an Existing Floating Scope or Scope Viewer”.

**Key Features**

- Multiple y-axes (displays) — Display multiple y-axes with multiple input ports. All the y-axes have a common time range on the x-axis.
- Multiple signals — Show multiple signals on the same y-axis (display) from one or more input ports.
- Modify parameters — Modify scope parameter values before and during a simulation.
- Display data after simulation — If a scope is closed at the start of a simulation, scope data is still written to the scope during the simulation. If you open the scope after a simulation, the scope displays simulation results for input signals.

**Oscilloscope Measurements**

- Triggers — Set triggers on repeating signals and pause the display when events occur.
- Cursor Measurements — Measure signal values using vertical and horizontal cursors.
- Signal Statistics\(^1\) — Display the maximum, minimum, peak-to-peak difference, mean, median, and RMS values of a selected signal.
- Peak Finder\(^a\) — Find maxima, showing the x-axis values at which they occur.
- Bilevel Measurements\(^a\) — Measure transitions, overshoots, undershoots, and cycles.

\(^1\) You must have a Simscape™ or DSP System Toolbox license to use the Peak Finder, Bilevel Measurements, and Signal Statistics.
Limitations

When you use model configuration parameters that optimize the simulation, such as Signal storage reuse or Block reduction, Simulink eliminates storage for some signals during simulation. You are unable to apply a Floating Scope to these eliminated signals. To work around this issue, configure an eliminated signal as a test point. You can then apply a Floating Scope to the signal regardless of optimization settings. To configure test points, see “Test Points”.

- If you step back the simulation after adding or removing a signal, the Floating Scope clears the existing data. New data does not appear until the simulation steps forward again.
- When connected to a constant signal, the scope plots a single point.
- Simulink messages are not supported for Floating Scope block and Scope Viewer.
- You cannot connect signals from ForEach subsystems.

Ports

Input

Port_1 — Signal or signals to visualize
scalar | vector | matrix | array | bus | nonvirtual bus

Connect the signals you want to visualize. You can have up to 96 input ports. Input signals can have these characteristics:

- **Type** — Continuous (sample-based) or discrete (sample-based and frame-based).
- **Data type** — Any data type that Simulink supports. See “Data Types Supported by Simulink”.
- **Dimension** — Scalar, one dimensional (vector), two dimensional (matrix), or multidimensional (array). Display multiple channels within one signal depending on the dimension. See “Signal Dimensions” and “Determine Output Signal Dimensions”.

Input Limitations

- When the input is a constant signal, the scope plots a single point.
- The scope shows gaps in the display when the signal value is NaN.
• When you visualize multiple frame-based signals in the scope, some samples of signals with a frame size of 1 might not be displayed. To visualize these signals, move the signals with frame size of 1 to a separate scope.

**Bus Support**

You can connect nonvirtual bus and arrays of bus signals to a scope. To display the bus signals, use normal or accelerator simulation mode. The scope displays each bus element signal in the order the elements appear in the bus, from the top to the bottom. Nested bus elements are flattened.

To log nonvirtual bus signals with a scope, set the *Save format* parameter to *Dataset*. You can use any *Save format* to log virtual bus signals.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `Boolean` | `fixed point` | `enumerated` | `bus`

### Properties

#### Configuration Properties

The Configuration Properties dialog box controls various properties about the scope displays. From the scope menu, select **View > Configuration Properties**.

**Main**

*Open at simulation start* — Specify when scope window opens

- off (default) | on

Select this check box to open the scope window when simulation starts.

**Programmatic Use**

See `OpenAtSimulationStart`.

*Display the full path* — Display block path on scope title bar

- off (default) | on

Select this check box to display the block path in addition to the block name.

**Layout** — Number and arrangement of displays

- 1-by-1 display (default) | an arrangement of *m*-by-*n* displays
Specify number and arrangement of displays. To expand the layout grid beyond 4 by 4, click within the dialog box and drag. The maximum layout is 16 rows by 16 columns.

![Diagram showing layout grid expansion](image)

If the number of displays is equal to the number of ports, signals from each port appear on separate displays. If the number of displays is less than the number of ports, signals from additional ports appear on the last display. For layouts with multiple columns and rows, ports are mapped down and then across.

**Programmatic Use**

See `LayoutDimensions`.

**Input processing — Channel or element signal processing**

Elements as channels (sample based) (default) | Columns as channels (frame based)

- Elements as channels (sample based) — Process each element as a unique sample.
- Columns as channels (frame based) — Process signal values in a column as a group of values from multiple time intervals. Frame-based processing is available only with discrete input signals.

**Programmatic Use**

See `FrameBasedProcessing`.

**Maximize axes — Maximize size of plots**

Off (default) | Auto | On
• **Auto** — If “Title” on page 1-0 and “Y-label” on page 1-0 properties are not specified, maximize all plots.

• **On** — Maximize all plots. Values in **Title** and **Y-label** are hidden.

• **Off** — Do not maximize plots.

**Programmatic Use**

See MaximizeAxes.

**Time**

**Time span** — Length of x-axis to display

*Auto (default) | User defined*

• **Auto** — Difference between the simulation start and stop times.

  The block calculates the beginning and end times of the time range using the “Time display offset” on page 1-0 and “Time span” on page 1-0 properties. For example, if you set the **Time display offset** to 10 and the **Time span** to 20, the scope sets the time range from 10 to 30.

• **User defined** — Enter any value less than the total simulation time.

**Programmatic Use**

See TimeSpan.

**Time span overrun action** — Display data beyond visible x-axis

*Wrap (default) | Scroll*

Specify how to display data beyond the visible x-axis range.

You can see the effects of this option only when plotting is slow with large models or small step sizes.

• **Wrap** — Draw a full screen of data from left to right, clear the screen, and then restart drawing the data from the left.

• **Scroll** — Move data to the left as new data is drawn on the right. This mode is graphically intensive and can affect run-time performance.

**Programmatic Use**

See TimeSpanOverrunAction.
**Time units — x-axis units**
None (default for Scope) | Metric (default for Time Scope) | Seconds

- Metric — Display time units based on the length of “Time span” on page 1-0.
- Seconds — Display time in seconds.
- None — Do not display time units.

**Programmatic Use**

See TimeUnits.

**Time display offset — x-axis offset**
0 (default) | scalar | vector

Offset the x-axis by a specified time value, specified as a real number or vector of real numbers.

For input signals with multiple channels, you can enter a scalar or vector:

- Scalar — Offset all channels of an input signal by the same time value.
- Vector — Independently offset the channels.

**Programmatic Use**

See TimeDisplayOffset.

**Time-axis labels — Display of x-axis labels**
Bottom Displays Only (default for Scope) | All (default for Time Scope) | None

Specify how x-axis (time) labels display:

- All — Display x-axis labels on all y-axes.
- None — Do not display labels. Selecting None also clears the Show time-axis label check box.
- Bottom displays only — Display x-axis label on the bottom y-axis.

**Dependencies**

To enable this property, set:

- “Show time-axis label” on page 1-0 to on.
• “Maximize axes” on page 1-0  to off.

The “Active display” on page 1-0  property determines which display is affected.

**Programmatic Use**

See TimeAxisLabels.

**Show time-axis label — Display or hide x-axis labels**

off (default for Scope) | on (default for Time Scope)

Select this check box to show the x-axis label for the active display

**Dependencies**

To enable this property, set “Time-axis labels” on page 1-0  to All or Bottom Displays Only.

The “Active display” on page 1-0  property determines which display is affected.

**Programmatic Use**

See ShowTimeAxisLabel.

**Display**

**Active display — Selected display**

1 (default) | positive integer

Selected display. Use this property to control which display is changed when changing style properties and axes-specific properties.

Specify the desired display using a positive integer that corresponds to the column-wise placement index. For layouts with multiple columns and rows, display numbers are mapped down and then across.

**Programmatic Use**

See ActiveDisplay.

**Title — Display name**

%<SignalLabel> (default) | string

Title for a display. The default value %<SignalLabel> uses the input signal name for the title.
**Dependency**

The “Active display” on page 1-0 property determines which display is affected.

**Programmatic Use**

See Title.

**Show legend — Display signal legend**

off (default) | on

Toggle signal legend. The names listed in the legend are the signal names from the model. For signals with multiple channels, a channel index is appended after the signal name. Continuous signals have straight lines before their names, and discrete signals have step-shaped lines.

From the legend, you can control which signals are visible. This control is equivalent to changing the visibility in the Style properties. In the scope legend, click a signal name to hide the signal in the scope. To show the signal, click the signal name again. To show only one signal, right-click the signal name, which hides all other signals. To show all signals, press Esc.

**Note** The legend only shows the first 20 signals. Any additional signals cannot be controlled from the legend.

**Dependency**

The “Active display” on page 1-0 property determines which display is affected.

**Programmatic Use**

See ShowLegend.

**Show grid — Show internal grid lines**

on (default) | off

Select this check box to show grid lines.

**Dependency**

The “Active display” on page 1-0 property determines which display is affected.
**Programmatic Use**

See ShowGrid.

**Plot signals as magnitude and phase — Split display into magnitude and phase plots**

off (default) | on

- On — Display magnitude and phase plots. If the signal is real, plots the absolute value of the signal for the magnitude. The phase is 0 degrees for positive values and 180 degrees for negative values. This feature is useful for complex-valued input signals. If the input is a real-valued signal, selecting this check box returns the absolute value of the signal for the magnitude.

- Off — Display signal plot. If the signal is complex, plots the real and imaginary parts on the same y-axis.

**Dependency**

The “Active display” on page 1-0 property determines which display is affected.

**Programmatic Use**

See PlotAsMagnitudePhase.

**Y-limits (Minimum) — Minimum y-axis value**

-10 (default) | real scalar

Specify the minimum value of the y-axis as a real number.

**Tunable:** Yes

**Dependency**

If you select *Plot signals as magnitude and phase*, this property only applies to the magnitude plot. The y-axis limits of the phase plot are always [-180 180].

The “Active display” on page 1-0 property determines which display is affected.

**Programmatic Use**

See YLimits.

**Y-limits (Maximum) — Maximum y-axis value**

10 (default) | real scalar
Specify the maximum value of the y-axis as a real number.

**Tunable:** Yes

**Dependency**

If you select **Plot signals as magnitude and phase**, this property only applies to the magnitude plot. The y-axis limits of the phase plot are always \([-180, 180]\).

The “Active display” on page 1-0 property determines which display is affected.

**Programmatic Use**

See `YLimits`.

**Y-label — Y-axis label**

none (default for Scope) | Amplitude (default for Time Scope) | string

Specify the text to display on the y-axis. To display signal units, add `(%<SignalUnits>)` to the label. At the beginning of a simulation, Simulink replaces `(%SignalUnits)` with the units associated with the signals.

Example: For a velocity signal with units of m/s, enter `Velocity (%<SignalUnits>)`.

**Dependency**

If you select **Plot signals as magnitude and phase**, this property does not apply. The y-axes are labeled Magnitude and Phase.

The “Active display” on page 1-0 property determines which display is affected.

**Programmatic Use**

See `YLabel`.

**Logging**

**Limit data points to last — Limit buffered data values**

off (default) | on

Limit buffered data values before plotting and saving signals. Data values are from the end of a simulation. To use this property, you must also specify the number of data values by entering a positive integer in the text box.
• On — Specify the number of data values saved for each signal (5000 by default). If the signal is frame-based, the number of buffered data values is the specified number of data values multiplied by the frame size.

For simulations with Stop time set to inf, consider selecting Limit data points to last.

Sometimes, selecting this parameter cause signals to be plotted for less than the entire time range of a simulation. For example, where the sample time is small. If a scope plots a portion of your signals, consider increasing the number of data values the simulation saves.

• Off — Save and plot all data values. Clearing Limit data points to last can cause an out-of-memory error for simulations that generate a large amount of data or for systems without enough available memory.

Dependency

To enable this property, select “Log data to workspace” on page 1-0.

This property limits the data values plotted in the scope and the data values saved to a MATLAB variable specified in “Variable name” on page 1-0.

Programmatic Use

See DataLoggingLimitDataPoints and DataLoggingMaxPoints.

Decimation — Reduce amount of scope data to display and save

off (default) | on

• On — Plot and log (save) scope data every Nth data point, where N is the decimation factor entered in the text box. The default decimation factor is 2. A value of 1 buffers all data values.

• Off — Save all scope data values.

Dependency

To enable this property, select “Log data to workspace” on page 1-0.

This property limits the data values plotted in the scope and the data values saved to a MATLAB variable specified in “Variable name” on page 1-0.
**Programmatic Use**

See `DataLoggingDecimateData` and `DataLoggingDecimation`.

**Log/Unlog Viewed Signals to Workspace — Toggle logging**
on | off

For signals selected with the Signal Selector, clicking this button toggles the state of the Log signal data check boxes in the Signals Properties dialog boxes.

**Axes Scaling Properties**

The Axes Scaling Properties dialog controls the axes limits of the scope. To open the Axes Scaling properties, in the scope menu, select **Tools > Axes Scaling > Axes Scaling Properties.**

**Axes scaling — Y-axis scaling mode**

- **Manual** (default) | **Auto** | **After N Updates**
  - Manual — Manually scale the y-axis range with the Scale Y-axis Limits toolbar button.
  - Auto — Scale the y-axis range during and after simulation. Selecting this option displays the “Do not allow Y-axis limits to shrink” on page 1-0 check box. If you want the y-axis range to increase and decrease with the maximum value of a signal, set Axes scaling to Auto and clear the Do not allow Y-axis limits to shrink check box.
  - After N Updates — Scale y-axis after the number of time steps specified in the “Number of updates” on page 1-0 text box (10 by default). Scaling occurs only once during each run.

**Programmatic Use**

See `AxesScaling`.

**Do not allow Y-axis limits to shrink — When y-axis limits can change**
on (default) | off

Allow y-axis range limits to increase but not decrease during a simulation.

**Dependency**

To use this property, set “Axes scaling” on page 1-0 to Auto.
Number of updates — Number of updates before scaling
10 (default) | integer

Set this property to delay auto scaling the y-axis.

Dependency

To use this property, set “Axes scaling” on page 1-0 to After N Updates.

Programmatic Use

See AxesScalingNumUpdates.

Scale axes limits at stop — When y-axis limits can change
on (default) | off

• On — Scale axes when simulation stops.
• Off — Scale axes continually.

Dependency

To use this property, set “Axes scaling” on page 1-0 to Auto.

Y-axis Data range (%) — Percent of y-axis to use for plotting
80 (default) | integer between [1, 100]

Specify the percentage of the y-axis range used for plotting data. If you set this property to 100, the plotted data uses the entire y-axis range.

Y-axis Align — Alignment along y-axis
Center (default) | Top | Bottom

Specify where to align plotted data along the y-axis data range when Y-axis Data range is set to less than 100 percent.

• Top — Align signals with the maximum values of the y-axis range.
• Center — Center signals between the minimum and maximum values.
• Bottom — Align signals with the minimum values of the y-axis range.

Autoscale X-axis limits — Scale x-axis range limits
off (default) | on
Scale x-axis range to fit all signal values. If **Axes scaling** is set to **Auto**, the data currently within the axes is scaled, not the entire signal in the data buffer.

**X-axis Data range (%) — Percent of x-axis to use for plotting**

100 (default) | integer in the range [1, 100]

Specify the percentage of the x-axis range to plot data on. For example, if you set this property to 100, plotted data uses the entire x-axis range.

**X-axis Align — Alignment along x-axis**

Center (default) | Top | Bottom

Specify where to align plotted data along the x-axis data range when **X-axis Data range** is set to less than 100 percent.

• **Top** — Align signals with the maximum values of the x-axis range.
• **Center** — Center signals between the minimum and maximum values.
• **Bottom** — Align signals with the minimum values of the x-axis range.

**Style Properties**

To open the Style dialog box, from the scope menu, select **View > Style**.

**Figure color — Background color for window**

black (default) | color

Background color for the scope.

**Plot type — How to plot signal**

Auto (default for Scope) | Line (default for Time Scope) | Stairs | Stem

When you select **Auto**, the plot type is a line graph for continuous signals, a stair-step graph for discrete signals, and a stem graph for Simulink message signals.

**Axes colors — Background and axes color for individual displays**

black (default) | color

Select the background color for axes (displays) with the first color palette. Select the grid and label color with the second color palette.
**Preserve colors for copy to clipboard — Copy scope without changing colors**
off (default) | on

Specify whether to use the displayed color of the scope when copying.

When you select **File > Copy to Clipboard**, the software changes the color of the scope to be printer friendly (white background, visible lines). If you want to copy and paste the scope with the colors displayed, select this check box.

**Properties for line — Line to change**
Channel 1 (default)

Select active line for setting line style properties.

**Visible — Line visibility**
on (default) | off

Show or hide a signal on the plot.

**Dependency**
The values of “Active display” on page 1-0 and “Properties for line” on page 1-0 determine which line is affected.

**Line — Line style**
solid line (default style) | 0.75 (default width) | yellow (default color)

Select line style, width, and color.

**Dependency**
The values of “Active display” on page 1-0 and “Properties for line” on page 1-0 determine which line is affected.

**Marker — Data point marker style**
None (default) | marker shape

Select marker shape.

**Dependency**
The values of “Active display” on page 1-0 and “Properties for line” on page 1-0 determine which line is affected.
### Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>double</th>
<th>enumerated</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Extended Capabilities

**C/C++ Code Generation**
Generate C and C++ code using Simulink® Coder™.

This block can be used for simulation visibility in systems that generate code, but is not included in the generated code.

**HDL Code Generation**
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

This block can be used for simulation visibility in subsystems that generate HDL code, but is not included in the hardware implementation.

**PLC Code Generation**
Generate Structured Text code using Simulink® PLC Coder™.

This block can be used for simulation visibility in systems that generate code, but is not included in the generated code.
Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

This block accepts fixed-point input, but converts it to double for display.

See Also

Blocks
Scope

Topics
“Scope Blocks and Scope Viewer Overview”
“Common Scope Block Tasks”
“Simulate a Model Interactively”
“Step Through a Simulation”
“Monitor Test Points in Stateflow Charts” (Stateflow)

Introduced in R2015b
For Each

Process elements or subarrays of a mask parameter or input signal independently

Library: Ports & Subsystems

Description

The For Each block serves as a control block for the For Each Subsystem block. Specifically, the For Each block enables the blocks inside the For Each Subsystem to process the elements of input signals or mask parameters independently. Each block inside this subsystem that has states maintains a separate set of states for each element or subarray that it processes. As the set of blocks in the subsystem processes the elements or subarrays, the subsystem concatenates the results to form output signals.

You can use a For Each subsystem to iteratively compute output after changing inputs or mask parameters. To do so, you configure the partitioning of input signals or mask parameters in the For Each block dialog box.

Partition Input Signals to the Subsystem

In a For Each subsystem, you can specify which input signals to partition for each iteration using the Input Partition tab in the dialog box of the For Each block. When specifying a signal to be partitioned, you also have to specify the Partition Dimension, Partition Width, and Partition Offset parameters.

Partition Parameters in the For Each block

You can partition the mask parameters of a For Each Subsystem block. Partitioning is useful for systems that have identical structures in each iteration but different parameter values. In this case, changing the model to partition extra input signals for each parameter is cumbersome. Instead, add a mask parameter to a For Each subsystem. For more information, see “Create a Simple Mask”. To select the mask parameter for...
partitioning, use the Parameter Partition tab on the For Each block dialog box. For more information, see “Select Partition Parameters” on page 1-809

**Concatenate Output**

You define the dimension along which to concatenate the results by specifying the **Concatenation Dimension** in the Output Concatenation tab.

The results generated by the block for each subarray stack along the concatenation dimension, \( d_1 \) (y-axis). Whereas, if you specify \( d_2 \) by setting the concatenation dimension to 2, the results concatenate along the \( d_2 \) direction (x-axis). Thus if the process generates row vectors, then the concatenated result is a row vector.

**Select Partition Parameters**

When selecting an input signal or subsystem mask parameter for partitioning, you need to specify how to decompose it into elements or subarrays for each iteration. Do this by setting integer values for the **Partition Dimension**, **Partition Width**, and **Partition Offset** parameters.

As an illustration, consider an input signal matrix \( A \) of the form:

\[
\begin{bmatrix}
A_{11} & A_{12} & A_{13} \\
A_{21} & A_{22} & A_{23} \\
A_{31} & A_{32} & A_{33}
\end{bmatrix}
\]

The labels \( d_1 \) and \( d_2 \), respectively, define dimensions 1 and 2. If you retain the default setting of 1 for both the partition dimension and the partition width, and 0 for the partition offset, then Simulink slices perpendicular to partition dimension \( d_1 \) at a width equal to the partition width, that is one element.
Matrix $A$ decomposes into these three row vectors.

$$
\begin{bmatrix}
A_{11} & A_{12} & A_{13} \\
A_{21} & A_{22} & A_{23} \\
A_{31} & A_{32} & A_{33}
\end{bmatrix}
$$

If instead you specify $d_2$ as the partition dimension by entering the value 2, Simulink slices perpendicular to $d_2$ to form three column vectors.

$$
\begin{bmatrix}
A_{11} & A_{12} & A_{13} \\
A_{21} & A_{22} & A_{23} \\
A_{31} & A_{32} & A_{33}
\end{bmatrix}
$$

In addition to setting the **Partition Dimension** to 2, if you set the **Partition Width** to 2 and the **Partition Offset** to -1, Simulink uses two overlapping $3x2$ partitions for processing.
For an example using the **Partition Offset** parameter, open the Simulink model `slexForEachOverlapExample`.

**Note** Only signals are considered one-dimensional in Simulink. Mask parameters are row or column vectors, according to their orientation. To partition a row vector, specify the partition dimension as 2 (along the columns). To partition a column vector, specify the partition dimension as 1 (along the rows).

**Ports**

**Input**

**In** — Signal input to a subsystem block  
scalar | vector | matrix

Placing an Inport block in a subsystem block adds an external input port to the block. The port label matches the name of the Inport block.

Use Inport blocks to get signals from the local environment.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus

**Output**

**Out** — Signal output from a subsystem  
scalar | vector | matrix

Placing an Outport block in a subsystem block adds an output port from the block. The port label on the subsystem block is the name of the Outport block.

Use Outport blocks to send signals to the local environment.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus
Parameters

Input Partition Tab

Select each input signal you want to partition and to specify the corresponding **Partition Dimension** and **Partition Width** parameters. See the Inport block reference page for more information.

**Port — List of input ports**

no default (default) | input port name

List of input ports connected to the For Each Subsystem block.

**Partition — Select input port signals to partition**

off (default) | on

Select input ports signals connected to the For Each Subsystem block to partition into subarrays or elements.

off

  Clear input port signals.

on

  Select input port signals to partition.

Dependency

Selecting this parameter enables the **Partition Dimension** and **Partition Width** parameters for the selected input port signal.

Programmatic Use

**Block Parameter:** InputPartition

**Type:** character vector

**Values:** 'off' | 'on'

**Default:** 'off'

**Partition Dimension — Specify dimension**

1 (default) | integer

Specify the dimension through which to slice the input signal array. The resulting slices are perpendicular to the dimension that you specify. The slices also partition the array into subarrays or elements, as appropriate.
1

Specify a dimension of 1.

integer

Specify dimension. Minimum value 1.

**Programmatic Use**

**Block Parameter:** InputPartitionDimension

**Type:** character vector

**Values:** '1' | '<integer>'

**Default:** '1'

**Partition Width — Specify width**

1 (default) | integer

Specify the width of each partition slice of the input signal.

1

Width of one element.

integer

Specify width. Minimum value 1.

**Programmatic Use**

**Block Parameter:** InputPartitionWidth

**Type:** character vector

**Values:** '1' | '<integer>'

**Default:** '1'

**Partition Offset — Specify partition offset**

0 (default) | integer

Specify the offset for each partition slice of the input signal.

0

No offset between partition slices.

integer

Specify partition offset where the sum of the partition width and the partition offset is a positive integer.

For example, a **Partition Width** of 3 and a **Partition Offset** of -2 indicates that each 3 element slice overlaps its neighboring slices by 2 elements.
Programmatic Use
Block Parameter: SubsysMaskParameterOffset
Type: character vector
Values: '0' | '<integer>'
Default: '0'

Show partition index output port (zero-based indexing) — Control display of output port
off (default) | on

Control display of output port for partition index.

off
  Hide output port.
on
  Display output port on block.

Dependency

Selecting this parameter enables the Partition index output data type parameter.

Programmatic Use
Block Parameter: ShowIterationIndex
Type: character vector
Values: 'off' | 'on'
Default: 'off'

Partition index output data type — Specify partition index data type
0 (default) | integer

Specify partition index data type for output port

int32
  Select signed 32-bit integer.
double
  Select double real.

Programmatic Use
Block Parameter: IterationIndexDataType
Type: character vector
Values: 'int32' | 'double'
Default: 'int32'

Output Concatenation Tab

For each output port, specify the dimension along which to stack (concatenate) the For Each Subsystem block results. See the Outport block reference page for more information.

Port — List of output ports
none (default) | output port name

List of output ports connected to the For Each Subsystem block.

Concatenation Dimension — Specify dimension
1 (default) | integer

Specify the dimension along which to stack the results of the For Each Subsystem block.

1
The results stack in the $d_1$ direction. If the block generates column vectors, the concatenation process results in a single column vector.

integer
The results stack in the $d_2$ direction. If the block generates row vectors, the concatenation process results in a single row vector. Minimum value 1

Programmatic Use
Block Parameter: OutputConcatenationDimension
Type: character vector
Values: '1' | '<integer>'
Default: '1'

Parameter Partition Tab

Select each mask parameter to partition and to specify the corresponding Partition Dimension and Partition Width parameters. Parameters appear in the list only if you have added an editable parameter to the mask of the parent For Each subsystem.

Parameter — List of mask parameters
parameter name
List of mask parameters for the For Each Subsystem block.

**Partition — Select mask parameters to partition**

| off (default) | on |

Select mask parameters for the For Each Subsystem block to partition into subarrays or elements.

- **off**
  - Clear mask parameters.
- **on**
  - Select mask parameters to partition.

**Dependency**

Selecting this parameter enables the **Partition Dimension** and **Partition Width** parameters for the selected mask parameter.

**Programmatic Use**

**Block Parameter:** `SubsysMaskParameterPartition`  
**Type:** character vector  
**Values:** 'off' | 'on'  
**Default:** 'off'

**Partition Dimension — Specify dimension**

| 1 (default) | integer |

Specify the dimension through which to slice the input signal array as an integer greater than or equal to one. The resulting slices are perpendicular to the dimension that you specify. The slices also partition the array into subarrays or elements, as appropriate.

**Programmatic Use**

**Block Parameter:** `SubsysMaskParameterPartitionDimension`  
**Type:** character vector  
**Values:** '1' | '<integer>'  
**Default:** '1'

**Partition Width — Specify partition width**

| 1 (default) | integer |

Specify the width of each partition slice of the input signal as an integer greater than or equal to one.
Programmatic Use

**Block Parameter:** SubsysMaskParameterPartitionWidth

**Type:** character vector

**Values:** '1' | '<integer>'

**Default:** '1'

See Also

**Blocks**

For Each Subsystem | Subsystem

**Topics**

“Repeat an Algorithm Using a For Each Subsystem”

“Log Signals in For Each Subsystems”

**Introduced in R2010a**
For Each Subsystem

Subsystem that repeats execution on each element or subarray of input signal and concatenates results

**Library:**
- Simulink / Ports & Subsystems
- HDL Coder / Ports & Subsystems

**Description**

The For Each Subsystem block is a Subsystem block preconfigured as a starting point for creating a subsystem that repeats execution during a simulation time step on each element or subarray of an input signal.

The set of blocks within the subsystem represents the algorithm applied to a single element or subarray of the original signal. The For Each block inside the subsystem allows you to configure the decomposition of the subsystem inputs into elements or subarrays, and to configure the concatenation of the individual results into output signals. The block parameters **Partition Dimension** and **Partition Width** specify the dimension through which to slice the input signal and the width of each slice respectively. To partition a row vector, specify the **Partition Dimension** as 2. To partition a column vector, specify the **Partition Dimension** as 1. To learn more about the block parameters, see For Each.
Inside this subsystem, each block that has states maintains separate sets of states for each element or subarray that it processes. Consequently, the operation of this subsystem is similar in behavior to copying the contents of the subsystem for each element in the original input signal and then processing each element using its respective copy of the subsystem.

An additional benefit of the For Each Subsystem block is that, for certain models, it improves the code reuse in SimulinkCoder generated code. Consider a model containing two reusable Atomic Subsystem blocks with the same scalar algorithm applied to each element of the signal. If the input signal dimensions of these subsystems are different, Simulink Coder generated code includes two distinct functions. You can replace these two subsystems with two identical For Each Subsystem blocks that are configured to process each element of their respective inputs using the same algorithm. For this case, Simulink Coder generated code consists of a single function parameterized by the number of input signal elements. This function is invoked twice — once for each unique instance of the For Each Subsystem block in the model. For each of these cases, the input signal elements have different values.

S-Function Support

The For Each Subsystem block supports both C-MEX S-functions and Level-2 MATLAB S-functions, provided that the S-function supports multiple execution instances using one of the following techniques:

- A C-MEX S-function must declare `ssSupportsMultipleExecInstances(S, true)` in the `mdlSetWorkWidths` method.
- A Level-2 MATLAB S-function must declare `block.SupportsMultipleExecInstances = true` in the setup method.

If you use the above specifications:

- Do not cache run-time data, such as DWork and Block I/O, using global or persistent variables or within the user data of the S-function.
- Every S-function execution method from `mdlStart` up to `mdlTerminate` is called once for each element processed by the S-function, when it is in a For Each Subsystem block. Consequently, you need to be careful not to free the same memory on repeated calls to `mdlTerminate`. For example, consider a C-MEX S-function that allocates memory for a run-time parameter within `mdlSetWorkWidths`. The memory only needs to be freed once in `mdlTerminate`. As a solution, set the pointer to be empty after the first call to `mdlTerminate`.
## Limitations

The For Each Subsystem block has these limitations, and these are the workarounds.

<table>
<thead>
<tr>
<th>Limitation</th>
<th>Workaround</th>
</tr>
</thead>
</table>
| You cannot log bus or an array of bus signals directly in the For Each subsystem. | Use one of these approaches:  
  • Use a Bus Selector block to select the signals you want to log and mark those signals for signal logging.  
  • Attach the signal to an Outport block and log the signal outside the For Each subsystem. |
| You cannot log a signal inside a referenced model that is inside a For Each subsystem if either of these conditions exists: | For the first condition, use Accelerator mode.  
  For the second condition, use Normal or Rapid Accelerator mode. |
  - The For Each subsystem is in a model simulating in Rapid Accelerator mode.  
  - The For Each subsystem itself is in a model referenced by a Model block in Accelerator mode. |
| You cannot log the states of the blocks in a For Each subsystem.          | Save and restore the simulation state.                                   |
| You cannot use Normal mode to simulate a Model block inside a For Each subsystem. | Use Accelerator or Rapid Accelerator mode.                               |
| Reusable code is generated for two For Each Subsystems with identical contents if their input and output signals are vectors (1-D or 2-D row or column vector). For n-D input and output signals, reusable code is generated only when the dimension along which the signal is partitioned is the highest dimension. | Permute the signal dimensions to transform the partition dimension and the concatenation dimension to the highest nonsingleton dimension for n-D signals. |

The For Each Subsystem block does not support these features:

- You cannot include these blocks or S-functions inside a For Each Subsystem:
• Data Store Memory, Data Store Read, or Data Store Write blocks inside the subsystem
• The From Workspace block if the input is a Structure with Time that has an empty time field
• The To Workspace and To File data saving blocks
• Goto and From blocks that cross the subsystem boundary
• Referenced model with simulation mode set to Normal
• Shadow Inports
• ERT S-functions

For a complete list of the blocks that support the For Each Subsystem, type `showblockdatatypetable` at the MATLAB command line.

• You cannot use these types of signals:
  • Test-pointed signals or signals with an external storage class inside the system
  • Frame signals on subsystem input and output boundaries
  • Variable-size signals
  • Function-call signals crossing the boundaries of the subsystem
• Creation of a linearization point inside the subsystem
• Propagating the Jacobian flag for the blocks inside the subsystem. You can check this condition in MATLAB using `J.Mi.BlockAnalyticFlags.jacobian`, where J is the Jacobian object. To verify the correctness of the Jacobian of the For Each Subsystem block, perform these steps:
  • Look at the tag of the For Each Subsystem Jacobian. If it is "not_supported", then the Jacobian is incorrect.
  • Move each block out of the For Each Subsystem and calculate its Jacobian. If any block is "not_supported" or has a warning tag, the For Each Subsystem Jacobian is incorrect.
• You cannot perform these types of code generation:
  • Generation of a Simulink Coder S-function target
  • Simulink Coder code generation under both of the following conditions:
    • A Stateflow or MATLAB Function block resides in the subsystem.
• This block tries to access global data outside the subsystem, such as Data Store Memory blocks or Simulink.Signal objects of ExportedGlobal storage class.
• PLC code generation

Ports

Input

In — Signal input to a subsystem block
scalar | vector | matrix

Placing an Inport block in a subsystem block adds an external input port to the block. The port label matches the name of the Inport block.

Use Inport blocks to get signals from the local environment.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus

Output

0ut — Signal output from a subsystem
scalar | vector | matrix

Placing an Outport block in a subsystem block adds an output port from the block. The port label on the subsystem block is the name of the Outport block.

Use Outport blocks to send signals to the local environment.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus
### Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>bus</th>
<th>double</th>
<th>enumerated</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>limited</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*a. Actual data type or capability support depends on block implementation.*

### Extended Capabilities

#### C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

Actual code generation support depends on block implementation.

#### HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

When you generate HDL code for the For Each Subsystem, the code generator uses a `for-generate` loop that iterates through elements of the input and output signals. The `for-generate` loop improves readability and reduces the number of lines of code, which can otherwise result in hundreds of lines of code for large vector signals.

You can use a nonzero partition offset for HDL code generation. The code generator supports partitioning of mask parameters inside a For Each Subsystem. Inside the For
Each Subsystem, use the mask parameter as the **Constant value** in Constant blocks or as the **Gain** parameter in Gain blocks.

**Limitations**

You cannot use the For Each Subsystem block as the DUT.

**HDL Architecture**

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Module (default)</td>
<td>Generate code for the subsystem and the blocks within the subsystem.</td>
</tr>
<tr>
<td>BlackBox</td>
<td>Generate a black box interface. The generated HDL code includes only the input/output port definitions for the subsystem. Therefore, you can use a subsystem in your model to generate an interface to existing, manually written HDL code. The black-box interface generation for subsystems is similar to the Model block interface generation without the clock signals.</td>
</tr>
<tr>
<td>No HDL</td>
<td>Remove the subsystem from the generated code. You can use the subsystem in simulation, however, treat it as a “no-op” in the HDL code.</td>
</tr>
</tbody>
</table>

**Black Box Interface Customization**

For the BlackBox architecture, you can customize port names and set attributes of the external component interface. See “Customize Black Box or HDL Cosimulation Interface” (HDL Coder).

**HDL Block Properties**

<table>
<thead>
<tr>
<th>General</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>AdaptivePipelining</strong></td>
</tr>
<tr>
<td><strong>BalanceDelays</strong></td>
</tr>
</tbody>
</table>
### General

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ClockRatePipelining</strong></td>
<td>Insert pipeline registers at a faster clock rate instead of the slower data rate. The default is inherit. See also “ClockRatePipelining” (HDL Coder).</td>
</tr>
<tr>
<td><strong>ConstrainedOutputPipeline</strong></td>
<td>Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. For more details, see “ConstrainedOutputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td><strong>DistributedPipelining</strong></td>
<td>Pipeline register distribution, or register retiming. The default is off. See also “DistributedPipelining” (HDL Coder).</td>
</tr>
<tr>
<td><strong>DSPStyle</strong></td>
<td>Synthesis attributes for multiplier mapping. The default is none. See also “DSPStyle” (HDL Coder).</td>
</tr>
<tr>
<td><strong>FlattenHierarchy</strong></td>
<td>Remove subsystem hierarchy from generated HDL code. The default is inherit. See also “FlattenHierarchy” (HDL Coder).</td>
</tr>
<tr>
<td><strong>InputPipeline</strong></td>
<td>Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “InputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td><strong>OutputPipeline</strong></td>
<td>Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “OutputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td><strong>SharingFactor</strong></td>
<td>Number of functionally equivalent resources to map to a single shared resource. The default is 0. See also “Resource Sharing” (HDL Coder).</td>
</tr>
<tr>
<td><strong>StreamingFactor</strong></td>
<td>Number of parallel data paths, or vectors, that are time multiplexed to transform into serial, scalar data paths. The default is 0, which implements fully parallel data paths. See also “Streaming” (HDL Coder).</td>
</tr>
</tbody>
</table>

### Target Specification

This block cannot be the DUT, so the block property settings in the **Target Specification** tab are ignored.
**Complex Data Support**

The block does not support complex data signals for HDL code generation. To input complex signals, you can convert this signal to an array of signals, and then input to the block. To learn more, see “Generate HDL Code for Blocks Inside For Each Subsystem” (HDL Coder).

**Fixed-Point Conversion**

Design and simulate fixed-point systems using Fixed-Point Designer™.

Actual data type support depends on block implementation.

**See Also**

- Blocks
  - For Each | Subsystem

**Topics**

- “Repeat an Algorithm Using a For Each Subsystem”
- “Log Signals in For Each Subsystems”

**Introduced in R2010a**
For Iterator

Repeat execution of a subsystem during a time step for a specified number of iterations
Library: Ports & Subsystems

Description

The For Iterator block, when placed in a Subsystem block, repeats the execution of a subsystem during the current time step until an iteration variable exceeds the specified iteration limit. You can use this block to implement the block diagram equivalent of a for loop in a programming language.

The output of a For Iterator Subsystem block cannot be a function-call signal. Simulink displays an error message when the model updates.

Ports

Input

Number of Iterations — External value for iterator variable
scalar | vector, size 1 | matrix, size 1x1

• The input port accepts data of mixed numeric types.
• If the input port value is non-integer, it is first truncated to an integer.
• Internally, the input value is cast to an integer of the type specified for the iteration variable output port.
• If no output port is specified, the input port value is cast to type int32.
• If the input port value exceeds the maximum value of the output port type, the overflow wraps around.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32
**Output**

**Iterator value — Value of iterator variable during time step**

scalar | vector, size 1 | matrix, size 1-by-1

Selecting the **Show iteration variable** parameter check box adds an output port to this block.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32

**Parameters**

**States when starting — Select block states between time steps**

hold (default) | reset

Select how to handle block states between time steps.

hold

Hold block states between time steps. Block state values persist across time steps.

reset

Reset block states to their initial values at the beginning of each time step and before the first iteration loop.

**Programmatic Use**

**Block Parameter:** ResetStates  
**Type:** character vector  
**Values:** 'hold' | 'reset'  
**Default:** 'hold'

**Iteration limit source — Select source for number of iterations**

internal (default) | external

Select source for number of iterations.

internal

Value of the **Iteration limit** parameter determines the number of iterations.

external

Value of the signal at the N port determines the number of iterations. The signal source must reside outside the For Iterator Subsystem block.
Dependencies

Selecting internal displays and enables the **Iteration limit** parameter. Selecting external adds an input port labeled N.

**Programmatic Use**

Block Parameter: IterationSource

Type: character vector

Values: 'internal' | 'external'

Default: 'internal'

**Iteration limit — Specify number of iterations**

5 (default) | integer

Specify the number of iterations. This parameter supports storage classes. You can define the named constant in the base workspace of the Model Explorer as a Simulink.Parameter object of the built-in storage class Define (custom) type.

5

Iterate blocks in the For Iterator Subsystem block 5 times.

integer

Specify an integer or a named constant variable.

Dependencies

To enable this parameter, select internal from the **Iteration limit source** drop-down list.

**Programmatic Use**

Block Parameter: IterationLimit

Type: character vector

Values: '5' | '<integer>'

Default: '5'

**Set Next i (iteration variable) externally — Control display of input port**

off (default) | on

Control display of an input port.

off

Remove input port.
on

Add input port labeled **Next_i** for connecting to an external iteration variable source. The value of the input at the current iteration is used as the value of the iteration variable at the next iteration.

**Dependencies**

To enable this parameter, select the **Show iteration variable** parameter which also displays an output port labeled **1:N**.

**Programmatic Use**

**Block Parameter:** ExternalIncrement  
**Type:** character vector  
**Values:** 'off' | 'on'  
**Default:** 'off'

**Show iteration variable — Control display of output port**

**on** (default) | **off**

Control the display of an output port with the current iterator value for a loop.

**on**

Add output port labeled **1:N** to the For Iterator block.

**off**

Remove output port.

**Dependencies**

Selecting this parameter enables the **Set next i (iteration variable) externally** parameter.

**Programmatic Use**

**Block Parameter:** ShowIterationPort  
**Type:** character vector  
**Value:** 'on' | 'off'  
**Default:** 'on'

**Index mode — Select initial iteration number**

**One-based** (default) | **Zero-based**

Select an initial iteration number of 0 or 1.
One-based
  Iteration number starts at one.

Zero-based
  Iteration number starts at zero.

**Programmatic Use**

**Block Parameter**: IndexMode  
**Type**: character vector  
**Values**: 'One-based' | 'Zero-based'  
**Default**: 'One-based'

**Iteration variable data type — Select data type**

`int32` (default) | `uint32` | `int16` | `uint16` | `int8` | `uint8` | `double`

Set the data type for the iteration value output from the iteration number port.

- `int32`
  Set data type to `int32`.
- `uint32`
  Set data type to `uint32`.
- `int16`
  Set data type to `int16`.
- `uint16`
  Set data type to `uint16`.
- `int8`
  Set data type to `int8`.
- `uint8`
  Set data type to `uint8`.
- `double`
  Set data type to `double`.

**Programmatic Use**

**Block Parameter**: IterationVariableDataType  
**Type**: character vector  
**Value**: 'int32' | 'uint32' | 'int16' | 'uint16' | 'int8' | 'uint8' | 'double'  
**Default**: 'int32'

See Also

Blocks
For Iterator Subsystem | Subsystem

Topics
Iterator Subsystem Execution

Introduced before R2006a
For Iterator Subsystem

Subsystem that repeats execution during a simulation time step

Library: Simulink / Ports & Subsystems

Description

The For Iterator Subsystem block is a Subsystem block preconfigured as a starting point for creating a subsystem that repeats the execution during a simulation time step for a specified number of iterations. The number of iterations is controlled by a For Iterator block.

When using simplified initialization mode, if you place a block that needs elapsed time (such as a Discrete-Time Integrator block) in a For Iterator Subsystem block, Simulink displays an error.
Ports

Input

In — Signal input to a subsystem block
scalar | vector | matrix

Placing an Inport block in a Subsystem block adds an external input port to the block. The port label matches the name of the Inport block.

Use Inport blocks to get signals from the local environment.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 |
Boolean | fixed point | enumerated | bus

Output

Out — Signal output from a subsystem
scalar | vector | matrix

Placing an Outport block in a Subsystem block adds an output port from the block. The port label on the Subsystem block is the name of the Outport block.

Use Outport blocks to send signals to the local environment.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 |
Boolean | fixed point | enumerated | bus

Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>bus</th>
<th>double</th>
<th>enumerated</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
<th>string</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>limited</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>limited(^a)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-----------------------</td>
<td>--------------</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\(^a\) Actual data type or capability support depends on block implementation.

### Extended Capabilities

#### C/C++ Code Generation
Generate C and C++ code using Simulink® Coder\(^\text{™} \). 
Actual data type or capability support depends on block implementation.

#### Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer\(^\text{™} \).
Actual data type or capability support depends on block implementation.

### See Also

**Blocks**
For Iterator | Subsystem

**Topics**
Iterator Subsystem Execution

**Introduced before R2006a**
From

Accept input from Goto block

Library: Simulink / Signal Routing
HDL Coder / Signal Routing

Description

The From block accepts a signal from a corresponding Goto block, then passes it as output. The data type of the output is the same as that of the input from the Goto block. From and Goto blocks allow you to pass a signal from one block to another without actually connecting them. To associate a Goto block with a From block, enter the Goto block's tag in the Goto Tag parameter.

A From block can receive its signal from only one Goto block, although a Goto block can pass its signal to more than one From block.

This figure shows that using a Goto block and a From block is equivalent to connecting the blocks to which those blocks are connected. In the model at the left, Block1 passes a signal to Block2. That model is equivalent to the model at the right, which connects Block1 to the Goto block, passes that signal to the From block, then on to Block2.

The visibility of a Goto block tag determines the From blocks that can receive its signal. For more information, see Goto and Goto Tag Visibility. The block indicates the visibility of the Goto block tag:

- A local tag name is enclosed in brackets ([ ]).
- A scoped tag name is enclosed in braces ({ }).
- A global tag name appears without additional characters.

The From block supports signal label propagation.
Ports

Output

Port_1 — Signal from connected Goto block
scalar | vector | matrix | N-D array

Signal from connected Goto block, output with the same dimensions and data type as the input to the Goto block.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus

Parameters

Goto Tag — Tag of the Goto block that forwards its signal to this block
A (default) | <More Tags...> | ...

Specify the tag of the Goto block that forwards its signal to this From block. To change the tag, select a new tag from the drop-down list.

The drop-down list displays the Goto tags that the From block can currently see. An item labeled <More Tags...> appears at the end of the list the first time you display the list in a Simulink session. Selecting this item causes the block to update the tags list to include the tags of Goto blocks residing in library subsystems referenced by the model containing this From block. Simulink software displays a progress bar while building the list of library tags. Simulink saves the updated tags list for the duration of the Simulink session or until the next time you select the adjacent Update Tags button. You need to update the tags list again in the current session only if the libraries referenced by the model have changed since the last time you updated the list.

Tip If you use multiple From and Goto Tag Visibility blocks to refer to the same Goto tag, you can simultaneously rename the tag in all of the blocks. To do so, use the Rename All button in the Goto block dialog box.

To find the relevant Goto block, use the Goto Source hyperlink in the From block dialog box.
Programmatic Use
Block Parameter: GotoTag
Type: character vector
Values: 'A' | ...
Default: 'A'

Update Tags — Update list of visible tags

Updates the list of tags visible to this From block, including tags residing in libraries referenced by the model containing this From block. Update the tags list again in the current session only if the libraries referenced by the model have changed since the last time you updated the list.

Goto Source — Path to connected Goto block
block path

Path of the Goto block connected to this From block. Clicking the path displays and highlights the Goto block in your model.

Icon Display — Text to display on block icon
Tag | Tag and signal name | Signal name

Specifies the text to display on the From block icon. The options are the block tag, the name of the signal that the block represents, or both the tag and the signal name.

Programmatic Use
Block Parameter: IconDisplay
Type: character vector
Values: 'Signal name' | 'Tag' | 'Tag and signal name'
Default: 'Tag'

Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>bus</th>
<th>double</th>
<th>enumerated</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
<th>string</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1-838
### Extended Capabilities

#### C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

#### HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

#### HDL Architecture
This block has a single, default HDL architecture.

#### HDL Block Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConstrainedOutputPipeline</td>
<td>Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. For more details, see “ConstrainedOutputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td>OutputPipeline</td>
<td>Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “OutputPipeline” (HDL Coder).</td>
</tr>
</tbody>
</table>
PLC Code Generation
Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

See Also
Goto | Goto Tag Visibility

Topics
“Signal Label Propagation”

Introduced before R2006a
From File

Load data from MAT-file
Library: Simulink / Sources

Description

The From File block loads data from a MAT-file to a model and outputs the data as a signal. The data is a sequence of samples. Each sample consists of a time stamp and an associated data value. The data can be in array format or MATLAB timeseries format.

The From File block icon shows the name of the MAT-file that supplies the data to the block.

You can have multiple From File blocks that load from the same MAT-file.

The supported MAT-file versions are Version 7.0 or earlier and Version 7.3. The From File block incrementally loads data from Version 7.3 files.

You can specify how the data is loaded, including:

- Sample time
- How to handle data for missing data points
- Whether to use zero-crossing detection

For more information, see “Load Data Using the From File Block”.

Ports

Output

Port 1 — File data
scalar | vector | matrix | N-D array
MAT-file data, specified as a sequence of samples. Each sample consists of a time stamp and an associated data value. The data can be in array format or MATLAB timeseries format.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus

Parameters

**File name — Path or file name**
untitled.mat (default) | path, or MAT-file name

Path or file name of the MAT-file that contains the input data. Specify a path or file name in one of these ways:

- Browse to a folder that contains a valid MAT-file.
  
  On UNIX® systems, the path name can start with a tilde (~) character, which means your home folder.
- Enter the path for the file in the text box.

The default file name is untitled.mat. If you specify a file name without path information, Simulink loads the file in the current folder or on the MATLAB path. (To determine the current folder, at the MATLAB command prompt enter `pwd`.)

After you specify the **File name**, you can use the view button ( ) to preview the signal from the MAT-file. For more information, see “Preview Signal Data”.

**Dependencies**

Code generation for RSIm target provides identical support as Simulink; all other code generation targets support only double, one-dimensional, real signals in array with time format.

To generate code that builds ERT or GRT targets or uses SIL or PIL simulation modes, the MAT-file must contain a nonempty, finite, real matrix with at least two rows.

For more information on C/C++ code generation with the From File block, see “Code Generation” on page 1-853.
Programmatic Use

**Block Parameter:** FileName

**Type:** character vector

**Values:** MAT-file name

**Default:** 'untitled.mat'

**Output data type — Output data type**

Inherit: auto (default) | double | single | int8 | uint8 | int16 | uint16 | int32 | uint32 | int64 | uint64 | boolean | fixdt(1,16,0) | fixdt(1,16,2^0,0) | Enum: <class_name> | Bus: <bus_object> | <data type expression>

The data type for the data that the From File block outputs. For nonbus types, you can use Inherit: auto to skip any data type verification. If you specify an output data type, then the From File block verifies that the data in the file matches the specified data type. For more information, see “Control Signal Data Types”.

If you set **Output data type** as a bus object, the bus object must be available when you compile the model. For each signal in bus data, the From File block verifies that data type, dimensions, and complexity are the same for the data and for the bus object.

Click the **Show data type assistant** button to display the Data Type Assistant, which helps you set the data type attributes. For more information, see “Specify Data Types Using Data Type Assistant”.

Programmatic Use

**Block Parameter:** OutDataTypeStr

**Type:** character vector

**Values:** 'Inherit: auto' | 'double' | 'single' | 'int8' | 'uint8' | 'int16' | 'uint16' | 'int32' | 'uint32' | 'int64' | 'uint64' | 'boolean' | 'fixdt(1,16,0)' | 'fixdt(1,16,2^0,0)' | 'Enum: <class name>' | 'Bus: <object name>' | '<data type expression>'

**Default:** 'Inherit: auto'

**Sample time — Sampling period and offset**

0 | scalar | vector

Specify the sample period and offset.

The From File block loads data from a MAT-file, using a sample time that either:

- You specify for the From File block.
- The From File block inherits from the blocks into which the From File block feeds data.

The default sample time is 0, which specifies a continuous sample time. The MAT-file is loaded at the base (fastest) rate of the model. For details, see “Specify Sample Time”.

**Programmatic Use**

**Block Parameter:** SampleTime

**Type:** character vector

**Values:** scalar | vector

**Default:** '0'

**Data extrapolation before first data point — Extrapolation method for simulation times before initial time stamp in MAT-file**

Linear extrapolation (default) | Hold first value | Ground value

Extrapolation method for a simulation time hit that occurs before the initial time stamp in the MAT-file. Choose one of the following extrapolation methods.
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear extrapolation</td>
<td>(Default)</td>
</tr>
<tr>
<td></td>
<td>If the MAT-file contains only one sample, then the From File block outputs the corresponding data value.</td>
</tr>
<tr>
<td></td>
<td>If the MAT-file contains more than one sample, then the From File block linearly extrapolates using the first two samples:</td>
</tr>
<tr>
<td></td>
<td>• For double data, linearly extrapolates the value using the first two samples</td>
</tr>
<tr>
<td></td>
<td>• For Boolean data, outputs the first data value</td>
</tr>
<tr>
<td></td>
<td>• For a built-in data type other than double or Boolean, the From File block:</td>
</tr>
<tr>
<td></td>
<td>• Upcasts the data to double</td>
</tr>
<tr>
<td></td>
<td>• Performs linear extrapolation (as described for double data)</td>
</tr>
<tr>
<td></td>
<td>• Downcasts the extrapolated data value to the original data type</td>
</tr>
<tr>
<td></td>
<td>You cannot use the Linear extrapolation option with enumerated (enum) data. All signals in a bus use the same extrapolation setting. If any signal in a bus uses enum data, then you cannot use the Linear extrapolation option.</td>
</tr>
<tr>
<td>Hold first value</td>
<td>Uses the first data value in the file</td>
</tr>
<tr>
<td>Ground value</td>
<td>Uses a value that depends on the data type of MAT-file sample data values:</td>
</tr>
<tr>
<td></td>
<td>• Fixed-point data types — Uses the ground value</td>
</tr>
<tr>
<td></td>
<td>• Numeric types other than fixed point — Uses 0</td>
</tr>
<tr>
<td></td>
<td>• Boolean — Uses false</td>
</tr>
<tr>
<td></td>
<td>• Enumerated data types — Uses default value</td>
</tr>
</tbody>
</table>
Dependencies

To generate code that builds ERT or GRT targets or uses SIL or PIL simulation modes, you must set this parameter to **Linear extrapolation**. For more information on C/C++ code generation with the From File block, see “Code Generation” on page 1-853.

**Programmatic Use**

**Block Parameter:** ExtrapolationBeforeFirstDataPoint  
**Type:** character vector  
**Values:** 'Linear extrapolation' | 'Hold first value' | 'Ground value'  
**Default:** 'Linear extrapolation'

**Data interpolation within time range — Interpolation method for simulation times that fall between two time stamps in the MAT-file**  
Linear interpolation (default) | Zero order hold

The interpolation method that Simulink uses for a simulation time hit between two time stamps in the MAT-file. Choose one of these interpolation methods.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear interpolation</td>
<td>(Default)</td>
</tr>
<tr>
<td></td>
<td>The From File block interpolates using the two corresponding MAT-file samples:</td>
</tr>
<tr>
<td></td>
<td>• For double data, linearly interpolates the value using the two corresponding samples</td>
</tr>
<tr>
<td></td>
<td>• For Boolean data, uses false for the first half of the sample and true for the second half.</td>
</tr>
<tr>
<td></td>
<td>• For a built-in data type other than double or Boolean, the From File block:</td>
</tr>
</tbody>
</table>
|                     | • Upcasts the data to double  
|                     | • Performs linear interpolation, as described for double data  
|                     | • Downcasts the interpolated value to the original data type |
| Zero order hold     | Uses the data from the first of the two samples                              |
Limitations

You cannot use the Linear interpolation option with enumerated (enum) data. All signals in a bus use the same interpolation setting. If any signal in a bus uses enum data, then you cannot use the Linear interpolation option.

Dependencies

To generate code that builds ERT or GRT targets or uses SIL or PIL simulation modes, you must set this parameter to Linear interpolation. For more information on C/C++ code generation with the From File block, see “Code Generation” on page 1-853.

Programmatic Use

**Block Parameter:** InterpolationWithinTimeRange  
**Type:** character vector  
**Values:** 'Linear interpolation' | 'Zero order hold'  
**Default:** 'Linear interpolation'

**Data extrapolation after last data point — Extrapolation method for simulation times after last time stamp in MAT-file**  
Linear extrapolation (default) | Hold last value | Ground value

The extrapolation method for a simulation time hit that occurs after the last time stamp in the MAT-file. Choose one of these extrapolation methods.
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear extrapolation</td>
<td>(Default)</td>
</tr>
<tr>
<td></td>
<td>If the MAT-file contains only one sample, then the From File block outputs the corresponding data value.</td>
</tr>
<tr>
<td></td>
<td>If the MAT-file contains more than one sample, then the From File block linearly extrapolates using data values of the last two samples:</td>
</tr>
<tr>
<td></td>
<td>• For double data, extrapolates the value using the last two samples.</td>
</tr>
<tr>
<td></td>
<td>• For Boolean data, outputs the first data value.</td>
</tr>
<tr>
<td></td>
<td>• For built-in data types other than double or Boolean:</td>
</tr>
<tr>
<td></td>
<td>• Upcasts the data to double</td>
</tr>
<tr>
<td></td>
<td>• Performs linear extrapolation, as described for double data</td>
</tr>
<tr>
<td></td>
<td>• Downcasts the extrapolated value to the original data type</td>
</tr>
<tr>
<td>Hold last value</td>
<td>Uses the last data value in the file</td>
</tr>
<tr>
<td>Ground value</td>
<td>Uses a value that depends on the data type of MAT-file sample data values:</td>
</tr>
<tr>
<td></td>
<td>• Fixed-point data types — Uses the ground value</td>
</tr>
<tr>
<td></td>
<td>• Numeric types other than fixed point — Uses 0</td>
</tr>
<tr>
<td></td>
<td>• Boolean — Uses false</td>
</tr>
<tr>
<td></td>
<td>• Enumerated data types — Uses default value</td>
</tr>
</tbody>
</table>

**Limitations**

You cannot use the Linear extrapolation option with enumerated (enum) data. All signals in a bus use the same extrapolation setting. If any signal in a bus uses enum data, then you cannot use the Linear extrapolation option.
**Dependencies**

To generate code that builds ERT or GRT targets or uses SIL or PIL simulation modes, you must set this parameter to *Linear extrapolation*. For more information on C/C++ code generation with the From File block, see “Code Generation” on page 1-853.

**Programmatic Use**

**Block Parameter:** ExtrapolationAfterLastDataPoint  
**Type:** character vector  
**Values:** 'Linear extrapolation' | 'Hold last value' | 'Ground value'  
**Default:** 'Linear extrapolation'

**Enable zero-crossing detection**  
**— Enable zero-crossing detection**  
off (default) | on

Enables zero-crossing detection.

The “Zero-Crossing Detection” parameter applies only if the **Sample time** parameter is set to 0 (continuous).

Simulink uses a technique known as zero-crossing detection to locate a discontinuity in time stamps, without resorting to excessively small time steps. “Zero-crossing” represents a discontinuity.

For the From File block, zero-crossing detection occurs only at time stamps in the file. Simulink examines only the time stamps, not the data values.

For bus signals, Simulink detects zero-crossings across all leaf bus elements.

If the input array contains duplicate time stamps (more than one entry with the same time stamp), Simulink detects a zero crossing at those time stamps. For example, suppose that the input array has this data.

```plaintext
| time:     | 0 1 2 2 3  
| signal:   | 2 3 4 5 6  
```

At time 2, there is a zero crossing from the input signal discontinuity.

For nonduplicate time stamps, zero-crossing detection depends on the settings of these parameters:

* **Data extrapolation before first data point**
• **Data interpolation within time range**
• **Data extrapolation after last data point**

The From File block determination of when zero-crossing occurs depends on the time stamp.

<table>
<thead>
<tr>
<th>Time Stamp</th>
<th>Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>First</td>
<td><strong>Data extrapolation before first data point</strong> is set to Ground value.</td>
</tr>
<tr>
<td>Between first and last</td>
<td><strong>Data interpolation within time range</strong> is set to Zero-order hold.</td>
</tr>
<tr>
<td>Last</td>
<td>One or both of these settings apply:</td>
</tr>
<tr>
<td></td>
<td>• <strong>Data extrapolation after last data point</strong> is set to Ground value.</td>
</tr>
<tr>
<td></td>
<td>• <strong>Data interpolation within time range</strong> is set to Zero-order hold.</td>
</tr>
</tbody>
</table>

This figure illustrates zero-crossing detection for data accessed by a From File block that has these settings:

• **Data extrapolation before first data point** — Linear extrapolation
• **Data interpolation within time range** (for internal points) — Zero order hold
• **Data extrapolation after last data point** — Linear extrapolation
This figure is another illustration of zero-crossing detection for data accessed by a From File block. The block has the following settings for the time stamps (points):

- **Data extrapolation before first data point** — Hold first value
- **Data interpolation within time range** — Zero order hold
- **Data extrapolation after last data point** — Hold last value
Dependencies

To generate code that builds ERT or GRT targets or uses SIL or PIL simulation modes, clear this check box. For more information on C/C++ code generation with the From File block, see “Code Generation” on page 1-853.

Programmatic Use

Block Parameter: ZeroCross
Type: character vector
Values: 'off' | 'on'
Default: 'on'
**Block Characteristics**

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>bus</th>
<th>double</th>
<th>enumerated</th>
<th>fixed point$^a$</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$^a$ Supports up to 32-bit fixed-point data types.

**Extended Capabilities**

**C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

- Not recommended for production code.
- Code generation for RSim target provides identical support as Simulink; all other code generation targets support only double, one-dimensional, real signals in array with time format.
- For a From File block, generating code that builds ERT or GRT targets or uses SIL or PIL simulation modes requires that:
  - The MAT-file contains a nonempty, finite, real matrix with at least two rows.
  - Use a data type of `double` for the matrix.
  - Do not include any NaN, Inf, or -Inf elements in the matrix.
  - In the From File block parameters dialog box:
    - Set the **Data extrapolation before first data point** and **Data extrapolation after last data point** parameters to **Linear extrapolation**.
• Set the **Data interpolation within time range** parameter to **Linear interpolation**.

• Clear the **Enable zero-crossing detection** parameter.

### Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

Supports up to 32-bit fixed-point data types.

### See Also
From Spreadsheet | From Workspace | To File | To Workspace

### Topics
“Overview of Signal Loading Techniques”
“Comparison of Signal Loading Techniques”
“Create Data for a From File Block”
“Load Data Using the From File Block”
“Load Signal Data That Uses Units”
“Specify Source for Data in Model Workspace”

**Introduced before R2006a**
From Spreadsheet

Read data from spreadsheet

Library: Simulink / Sources

Description

The From Spreadsheet block reads data from Microsoft Excel® (all platforms) or CSV (MicrosoftWindows® platform with Microsoft Office installed only) spreadsheets and outputs the data as a signal. The From Spreadsheet block does not support Microsoft Excel spreadsheet charts.

The From Spreadsheet icon displays the spreadsheet file name and sheet name specified in the block File name and Sheet name parameters.

Storage Formats

The data that the From Spreadsheet block reads from a spreadsheet must be appropriately formatted.

For Microsoft Excel spreadsheets:

- The From Spreadsheet block interprets the first row as a signal name. If you do not specify a signal name, the From Spreadsheet block assigns a default one with the format Signal #, where # increments with each additional unnamed signal.
- The From Spreadsheet block interprets the first column as time. In this column, the time values must monotonically increase.
- The From Spreadsheet block interprets the remaining columns as signals.

This example shows an acceptably formatted Microsoft Excel spreadsheet. The first column is Time and the first row contains signal names. Each worksheet contains a signal group.
For CSV text files (Microsoft platform with Microsoft Office installed only):

- The From Spreadsheet block interprets the first column as time. In this column, the time values must increase.
- The From Spreadsheet block interprets the remaining columns as signals.
- Each column must have the same number of entries.
- The From Spreadsheet block interprets each file as one signal group.

This example shows an acceptably formatted CSV file. The contents represent one signal group.

0,0,0,5,0
1,0,1,5,0
2,0,1,5,0
3,0,1,5,0
4,5,1,5,0
Block Behavior During Simulation

The From Spreadsheet block incrementally reads data from the spreadsheet during simulation.

The Sample time parameter specifies the sample time that the From Spreadsheet block uses to read data from the spreadsheet. For details, see “Parameters” on page 1-859. The time stamps in the file must be monotonically nondecreasing.

For each simulation time hit for which the spreadsheet contains no matching time stamp, Simulink software interpolates or extrapolates to obtain the needed data using the selected method. For details, see “Simulation Time Hits That Have No Corresponding Spreadsheet Time Stamps” on page 1-857.

Simulation Time Hits That Have No Corresponding Spreadsheet Time Stamps

If the simulation time hit does not have a corresponding spreadsheet time stamp, the From Spreadsheet block output depends on:

- Whether the simulation time hit occurs before the first time stamp, within the range of time stamps, or after the last time stamp
- The interpolation or extrapolation methods that you select
- The data type of the spreadsheet data

For details about interpolation and extrapolation options, see the descriptions of these parameters:

- “Data extrapolation before first data point” on page 1-0
- “Data interpolation within time range” on page 1-0
- “Data extrapolation after last data point” on page 1-0
Sometimes the spreadsheet includes two or more data values that have the same time stamp. In such cases, the From Spreadsheet block action depends on when the simulation time hit occurs, relative to the duplicate time stamps in the spreadsheet.

For example, suppose that the spreadsheet contains this data. Three data values have a time stamp value of 2.

time stamps:    0 1 2 2 2 3 4  
data values:  2 3 6 4 9 1 5

The table describes the From Spreadsheet block output.

<table>
<thead>
<tr>
<th>Simulation Time, Relative to Duplicate Time Stamp Values in Spreadsheet</th>
<th>From Spreadsheet Block Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Before the duplicate time stamps</td>
<td>Performs the same actions as when the time stamps are distinct, using the first of the duplicate time stamp values as the basis for interpolation. (In this example, the time stamp value is 6.)</td>
</tr>
<tr>
<td>At or after the duplicate time stamps</td>
<td>Performs the same actions as when the times stamps are distinct, using the last of the duplicate time stamp values as the basis for interpolation. (In this example, that time stamp value is 9.)</td>
</tr>
</tbody>
</table>

**Ports**

**Output**

**Port_1 — Data from spreadsheet**

scalar | vector | matrix

Incremental data from the specified spreadsheet.

The **Sample time** parameter specifies the sample time that the From Spreadsheet block uses to read data from the spreadsheet. For details, see “Parameters” on page 1-859. The time stamps in the file must be monotonically nondecreasing.

For each simulation time hit for which the spreadsheet contains no matching time stamp, Simulink software interpolates or extrapolates to obtain the needed data using the
selected method. For details, see “Simulation Time Hits That Have No Corresponding Spreadsheet Time Stamps” on page 1-857.

The From Spreadsheet block accepts data type specifications at a block level. If you want to specify different data types for each signal, consider selecting **Output Data Type** > **Inherit: Auto**. This option resolves back signal data types using back propagation. For example, assume that there are two signals in the From Spreadsheet block, In1 and In2, which the block sends to ports that have int8 and Boolean data types. With back propagation, the block recasts In1 as int8 and In2 as Boolean.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated

**Parameters**

**File name — Full path and file name**

*untitled.xlsx* (default) | full path and file name

Enter full path and file name of a spreadsheet file.

This block supports non-English full paths and file names only on Microsoft platforms.

**Programmatic Use**

**Block Parameter:** FileName

**Type:** character vector

**Value:** full path and file name

**Default:** 'untitled.xlsx'

**Sheet name — Name of sheet in spreadsheet**

*Sheet1* (default) | sheet name

Enter the name of the sheet in the spreadsheet. You can type the sheet name in this edit box or select the sheet name after you open the sheet.

If your spreadsheet is the CSV format, the block populates this parameter with the name of the CSV file without the extension. Do not change this value.

To open the sheet, click ![Open Sheet]. In the sheet, you can select the range of data by dragging over the desired range of values.
Alternatively, you can select the range of data by specifying the range of values in the Range parameter.

**Programmatic Use**

**Block Parameter:** SheetName  
**Type:** character vector  
**Value:** Sheet name  
**Default:** 'Sheet1'

**Range — Cell range**

entire range of used cells in sheet (default) | A1:B3,D1:D3,A7:B9,D7:D9 | comma-separated list of column:row

To specify the range, use the format column:row, with multiple specifications separated by commas. For example, A1:B3,D1:D3,A7:B9,D7:D9. If unspecified, or empty, the block automatically detects the used range, which is all the data in the sheet.
If the selections overlap, the block resolves the selection information as appropriate. For example, if you specify multiple ranges that overlap, such as A1:B4,B1:E7, the block resolves the selection to A1 to E7, inclusive.

An alternate to using the **Range** parameter is to open the sheet, by clicking . In the sheet, you can select the range of data by dragging over the desired range of values.

**Programmatic Use**

**Block Parameter:** Range  
**Type:** character vector  
**Value:** Cell range  
**Default:** ''

**Output data type — Output data type**

Inherit: auto (default) | double | single | int8 | uint8 | int16 | uint16 | int32 | uint32 | int64 | uint64 | boolean | fixdt(1,16,0) | fixdt(1,16,2^0,0) | Enum: <class_name> | Bus: <bus_object> | <data type expression>

The data type for the From Spreadsheet block output. The From Spreadsheet block accepts spreadsheets that contain many data types. However, the block reads the spreadsheet data type as doubles. It then outputs the data type according to the value of **Output data type**.

If you want to specify different data types for each signal, consider selecting **Output Data Type > Inherit: auto**. This option resolves back signal data types using back propagation. For example, assume that there are two signals in the From Spreadsheet block, In1 and In2, which the block sends to ports that have int8 and Boolean data types. With back propagation, the block recasts In1 as int8 and In2 as boolean.
To allow the block to cast the output data type to match that of the receiving block, use `Inherit: auto`.

For more information, see “Control Signal Data Types”.

**Programmatic Use**

**Block Parameter:** OutDataTypeStr

**Type:** character vector

**Values:** 'Inherit: auto' | 'double' | 'single' | 'int8' | 'uint8' | 'int16' | 'uint16' | 'int32' | 'uint32' | 'int64' | 'uint64' | 'boolean' | 'fixdt(1,16,0)' | 'fixdt(1,16,2^0,0)' | 'Enum: <class name>' | 'Bus: <object name>' | '<data type expression>'

**Default:** 'Inherit: auto'

**Treat first column as — Time or data**

Time (default) | Data

Select how the block should treat the first column of the spreadsheet:

- **Time** — Treat first column as time.
- **Data** — Treat first column as data.

**Dependencies**

When you select **Data**, the block disables:

- **Data extrapolation before first data point**
- **Data interpolation within time range**
- **Data extrapolation after last data point**

And enables:

- **Output after last data point**

**Programmatic Use**

**Block Parameter:** TreatFirstColumnAs

**Type:** character vector

**Value:** 'Time' | 'Data'

**Default:** 'Time'

**Sample time — Sampling period and offset**

0 (default) | scalar | vector
The sample period and offset.

The From Spreadsheet block reads data from a spreadsheet using a sample time that either:

- You specify for the From Spreadsheet block
- The From Spreadsheet block inherits from the blocks into which the From Spreadsheet block feeds data

The default is 0, which specifies a continuous sample time. The spreadsheet is read at the base (fastest) rate of the model. For details, see “Specify Sample Time”.

**Programmatic Use**

**Parameter:** SampleTime  
**Type:** character vector  
**Values:** scalar | vector  
**Default:** '0'

**Data extrapolation before first data point — Extrapolation method for simulation times before initial time stamp in MAT-file**  
Linear extrapolation (default) | Hold first value | Ground value

Extrapolation method that Simulink uses for a simulation time hit that is before the first time stamp in the spreadsheet. Choose one of these extrapolation methods.
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear extrapolation</td>
<td>(Default)</td>
</tr>
<tr>
<td></td>
<td>If the spreadsheet contains only one sample, the From Spreadsheet block outputs the corresponding data value.</td>
</tr>
<tr>
<td></td>
<td>If the spreadsheet contains more than one sample, the From Spreadsheet block linearly extrapolates using the first two samples:</td>
</tr>
<tr>
<td></td>
<td>• For double data, linearly extrapolates the value using the first two samples</td>
</tr>
<tr>
<td></td>
<td>• For Boolean data, outputs the first data value</td>
</tr>
<tr>
<td></td>
<td>• For a built-in data type other than double or Boolean:</td>
</tr>
<tr>
<td></td>
<td>• Upcasts the data to double</td>
</tr>
<tr>
<td></td>
<td>• Performs linear extrapolation (as described above for double data)</td>
</tr>
<tr>
<td></td>
<td>• Downcasts the extrapolated data value to the original data type</td>
</tr>
<tr>
<td></td>
<td>You cannot use the Linear extrapolation option with enumerated (enum) data.</td>
</tr>
<tr>
<td>Hold first value</td>
<td>Uses the first data value in the file</td>
</tr>
<tr>
<td>Ground value</td>
<td>Uses a value that depends on the data type of spreadsheet sample data values:</td>
</tr>
<tr>
<td></td>
<td>• Fixed-point data types — Uses the ground value</td>
</tr>
<tr>
<td></td>
<td>• Numeric types other than fixed-point — Uses 0</td>
</tr>
<tr>
<td></td>
<td>• Boolean — Uses false</td>
</tr>
<tr>
<td></td>
<td>• Enumerated data types — Uses default value</td>
</tr>
</tbody>
</table>

**Dependencies**

To enable this parameter, set **Treat first column as** to **Time**.

**Programmatic Use**

**Parameter:** ExtrapolationBeforeFirstDataPoint
Type: character vector
Values: 'Linear extrapolation' | 'Hold first value' | 'Ground value'
Default: 'Linear extrapolation'

**Data interpolation within time range — Interpolation method for simulation times that fall between two time stamps in the MAT-file**

The interpolation method that Simulink uses for a simulation time hit between two time stamps in the spreadsheet. Choose one of the following interpolation methods.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear interpolation</td>
<td>(Default)</td>
</tr>
<tr>
<td></td>
<td>The From Spreadsheet block interpolates using the two corresponding spreadsheet samples:</td>
</tr>
<tr>
<td></td>
<td>• For double data, linearly interpolates the value using the two corresponding samples</td>
</tr>
<tr>
<td></td>
<td>• For Boolean data, uses false for the first half of the sample and true for the second half</td>
</tr>
<tr>
<td></td>
<td>• For a built-in data type other than double or Boolean:</td>
</tr>
<tr>
<td></td>
<td>• Upcasts the data to double</td>
</tr>
<tr>
<td></td>
<td>• Performs linear interpolation (as described above for double data)</td>
</tr>
<tr>
<td></td>
<td>• Downcasts the interpolated value to the original data type</td>
</tr>
<tr>
<td></td>
<td>You cannot use the Linear interpolation option with enumerated (enum) data.</td>
</tr>
<tr>
<td>Zero order hold</td>
<td>Uses the data from the first of the two samples</td>
</tr>
</tbody>
</table>

**Dependencies**

To enable this parameter, set **Treat first column as** to Time.

**Programmatic Use**

**Parameter:** InterpolationWithinTimeRange  
**Type:** character vector
**Values:** 'Linear interpolation' | 'Zero order hold'

**Default:** 'Linear interpolation'

**Data extrapolation after last data point — Extrapolation method for simulation times after last time stamp in MAT-file**

Linear extrapolation (default) | Hold last value | Ground value

The extrapolation method that Simulink uses for a simulation time hit that is after the last time stamp in the spreadsheet. Choose one of the following extrapolation methods.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear extrapolation</td>
<td>(Default)</td>
</tr>
<tr>
<td></td>
<td>If the spreadsheet contains only one sample, the From Spreadsheet block outputs the corresponding data value.</td>
</tr>
<tr>
<td></td>
<td>If the spreadsheet contains more than one sample, the From Spreadsheet block linearly extrapolates using data values of the last two samples:</td>
</tr>
<tr>
<td></td>
<td>• For double data, extrapolates the value using the last two samples</td>
</tr>
<tr>
<td></td>
<td>• For Boolean data, outputs the last data value</td>
</tr>
<tr>
<td></td>
<td>• For a built-in data type other than double or Boolean:</td>
</tr>
<tr>
<td></td>
<td>• Upcasts the data to double.</td>
</tr>
<tr>
<td></td>
<td>• Performs linear extrapolation (as described above for double data).</td>
</tr>
<tr>
<td></td>
<td>• Downcasts the extrapolated value to the original data type.</td>
</tr>
<tr>
<td></td>
<td>You cannot use the Linear extrapolation option with enumerated (enum) data.</td>
</tr>
<tr>
<td>Hold last value</td>
<td>Uses the last data value in the file</td>
</tr>
</tbody>
</table>
Method | Description
--- | ---
Ground value | Uses a value that depends on the data type of spreadsheet sample data values:
- Fixed-point data types — Uses the ground value
- Numeric types other than fixed-point — uses 0
- Boolean — Uses false
- Enumerated data types — Uses default value

Dependencies

To enable this parameter, set Treat first column as to Time.

**Programmatic Use**
**Parameter:** ExtrapolationAfterLastDataPoint
**Type:** character vector
**Values:** 'Linear extrapolation' | 'Hold last value' | 'Ground value'
**Default:** 'Linear extrapolation'

**Output after last data point — Action after last data point**
Repeating sequence (default) | Hold final value | Ground value

Select action after last data point:

- **Repeating sequence** — Repeat the sequence by reading the data from the first row of the range specified in Range
- **Hold final value** — Output the last defined value for the remainder of the simulation.
- **Ground value** — Output a ground value depending on the data type value specified in Output data type.

Dependencies

To enable this parameter, set Treat first column as to Data.

**Programmatic Use**
**Parameter:** OutputAfterLastPoint
**Type:** character vector
**Values:** 'Repeating sequence' | 'Hold final value' | 'Ground value'
**Default:** 'Repeating sequence'
Enable zero-crossing detection — Enable zero-crossing detection

off (default) | on

Select to enable zero-crossing detection.

The “Zero-Crossing Detection” parameter applies only if the Sample time parameter is set to $0$ (continuous).

Simulink uses a technique known as zero-crossing detection to locate accurately a discontinuity, without resorting to excessively small time steps. In this context, zero-crossing is used to represent discontinuities.

For the From Spreadsheet block, zero-crossing detection can only occur at time stamps in the file. Simulink examines only the time stamps, not the data values.

If the input array contains duplicate time stamps (more than one entry with the same time stamp), Simulink detects a zero crossing at that time stamp. For example, suppose that the input array has this data.

```
time:     0 1 2 2 3
signal:   2 3 4 5 6
```

At time 2, there is a zero crossing from the input signal discontinuity.

For data with nonduplicate time stamps, zero-crossing detection depends on the settings of the following parameters:

- **Data extrapolation before first data point**
- **Data interpolation within time range**
- **Data extrapolation after last data point**

The block applies the following rules when determining when:

- Zero-crossing occurs for the first time stamp
- For time stamps between the first and last time stamp
- For the last time stamp

<table>
<thead>
<tr>
<th>Time Stamp</th>
<th>When Zero-Crossing Detection Occurs</th>
</tr>
</thead>
<tbody>
<tr>
<td>First</td>
<td>Data extrapolation before first data point is set to Ground value.</td>
</tr>
</tbody>
</table>

1-868
<table>
<thead>
<tr>
<th>Time Stamp</th>
<th>When Zero-Crossing Detection Occurs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Between first and last</td>
<td><strong>Data interpolation within time range</strong> is set to <strong>Zero-order hold</strong>.</td>
</tr>
<tr>
<td>Last</td>
<td>One or both of these settings occur:</td>
</tr>
<tr>
<td></td>
<td>• <strong>Data extrapolation after last data point</strong> is set to <strong>Ground value</strong>.</td>
</tr>
<tr>
<td></td>
<td>• <strong>Data interpolation within time range</strong> is set to <strong>Zero-order hold</strong>.</td>
</tr>
</tbody>
</table>

The following figure illustrates zero-crossing detection for data accessed by a From Spreadsheet block that has these settings:

- **Data extrapolation before first data point** — **Linear extrapolation**
- **Data interpolation within time range** (for internal points) — **Zero order hold**
- **Data extrapolation after last data point** — **Linear extrapolation**
The following figure is another illustration of zero-crossing detection for data accessed by a From Spreadsheet block. The block has these settings for the time stamps (points):

- **Data extrapolation before first data point** — Hold first value
- **Data interpolation within time range** — Zero order hold
- **Data extrapolation after last data point** — Hold last value
Programmatic Use
Block Parameter: ZeroCross
Type: character vector
Values: 'off' | 'on'
Default: 'off'

Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>double</th>
<th>enumerated</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1-871
### Algorithms

When the From Spreadsheet block reads data from a spreadsheet, it reads all signals with double precision and then casts them to the data type specified in the **Output data type** parameter for the output signal. During casting, the block uses rounding mode and saturation on integer overflow as follows.

#### Rounding Mode

The From Spreadsheet block rounds positive and negative numbers toward negative infinity. This mode is equivalent to the MATLAB `floor` function.

#### Saturation on Integer Overflow

For data type conversion, the From Spreadsheet block deals with saturation overflow by wrapping to the appropriate value that the data type can represent. For example, the number 130 does not fit in a signed 8-bit integer and wraps to –126.

### Extended Capabilities

**C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

- Not recommended for production code.
• Code generation for RSim target provides identical support as Simulink; all other code generation targets support only double, one-dimensional, real signals in array with time format.

• Simulating in accelerator, rapid accelerator, model reference accelerator mode, or model reference rapid accelerator mode behaves the same way, and has the same requirements, as simulating in normal mode.

• The From Spreadsheet block does not support generating code that involves building ERT or GRT targets, or using SIL or PIL simulation modes.

**Fixed-Point Conversion**
Design and simulate fixed-point systems using Fixed-Point Designer™.

Supports up to 32-bit fixed-point data types.

**See Also**
From File | From Workspace

**Topics**
“Overview of Signal Loading Techniques”
“Comparison of Signal Loading Techniques”
“Spreadsheets” (MATLAB)

**Introduced in R2015b**
From Workspace

Load signal data from workspace
Library: Simulink / Sources

Description

The From Workspace block reads signal data from a workspace and outputs the data as a signal.

The block displays the expression specified in the Data parameter. For details about how Simulink software evaluates this expression, see “Symbol Resolution”.

You can specify how the data is loaded, including sample time, how to handle data for missing data points, and whether to use zero-crossing detection. For more information, see “Load Data Using the From Workspace Block”.

Note Simulink Coder software does not generate code for this block when connected to the Sim port under the conditions listed in Environment Controller.

Specifying Workspace Data

In the From Workspace block dialog box, use the Data parameter to specify the workspace data to load. You can specify a MATLAB expression (for example, the name of a variable in the MATLAB workspace) that evaluates to one of these options:

- A MATLAB timeseries object
- A structure of MATLAB timeseries objects
- A structure, with or without time
- A two-dimensional matrix

For additional information, see “Load Data Using the From Workspace Block”.
**Use with Data Dictionary**

When you link a model to a data dictionary, you:

- Store design data, which contributes to the fundamental design of the model in the Design Data section of the dictionary. Design data includes numeric variables and Simulink.Parameter objects that you use to set block parameter values.
- Store simulation input data, which you use to stimulate and experiment with the model, in the base workspace. Typically, you create simulation input data as MATLAB timeseries objects.

For more information about storing variables, objects, and other data that a model uses, see “Determine Where to Store Variables and Objects for Simulink Models”.

- To access design data by using a From Workspace block, store the target variable in the Design Data section of the dictionary and set the Data parameter of the block to the name of the variable.
- To access simulation input data, store the target variable in the base workspace and set the Data parameter by using a call to the evalin function. In the call to evalin, specify the ws argument as 'base' so that the block seeks the variable in the base workspace instead of the data dictionary. For example, if the name of the variable is myTimeseriesObject, set Data to evalin('base','myTimeseriesObject').

**Ports**

**Output**

**Port_1 — Workspace data**

scalar | vector | matrix

Signal created from workspace data. The block outputs real or complex signals of any type that Simulink supports, including fixed-point and enumerated data types.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus
Parameters

Data — Workspace data to load

simin (default) | timeseries object | structure of timeseries objects | structure | 2-D matrix

In the Data parameter, specify the workspace data to load. Specify a MATLAB expression (for example, the name of a variable in the MATLAB workspace) that evaluates to one of the following:

- A MATLAB timeseries object
- A structure of MATLAB timeseries objects
- A structure, with or without time
- A two-dimensional matrix

The From Workspace block also accepts a bus object as a data type. To load bus data, use a structure of MATLAB timeseries objects. For details, see “Load Bus Data to Root-Level Input Ports”.

Real signals of type double can be in any data format that the From Workspace block supports. For complex signals and real signals of a data type other than double, use any format except Array.

For additional information, see “Specify the Workspace Data”.

Programmatic Use

Block Parameter: VariableName
Type: character vector
Values: timeseries object | structure of timeseries objects | structure | 2-D matrix
Default: 'simin'

Output data type — Output data type

Inherit: auto (default) | double | single | int8 | uint8 | int16 | uint16 | int32 | uint32 | int64 | uint64 | boolean | fixdt(1,16,0) | fixdt(1,16,2^0,0) | Enum: <class_name> | Bus: <bus_object> | <data type expression>

Required data type for the workspace data that the From Workspace block loads. For non-bus types, to skip any data type verification, you can use Inherit: auto. For more information, see “Control Signal Data Types”.

1-876
To load bus data, use a structure of MATLAB timeseries objects. For details, see “Load Bus Data to Root-Level Input Ports”.

**Programmatic Use**

**Block Parameter:** OutDataTypeStr  
**Type:** character vector  
**Values:**  
'Inherit: auto' | 'double' | 'single' | 'int8' | 'uint8' |  
'int16' | 'uint16' | 'int32' | 'uint32' | 'int64' | 'uint64' |  
'boolean' | 'fixdt(1,16,0)' | 'fixdt(1,16,2^0,0)' | 'Enum: <class name>' | 'Bus: <object name>' | '<data type expression>'  
**Default:** 'Inherit: auto'

**Sample time — Sample rate of loaded data**

0 (default) | scalar | vector

Sample rate of loaded workspace data. For details, see “Specify Sample Time”.

**Command-Line Information**

**Block Parameter:** SampleTime  
**Type:** character vector  
**Values:** scalar | vector  
**Default:** '0'

**Interpolate data — Interpolate or use last known data point**

on (default) | off

When you select this option, the block performs linear interpolation at time hits for which no corresponding workspace data exist. Otherwise, the current output equals the output at the most recent time for which data exists.

The From Workspace block interpolates by using the two corresponding workspace samples:

- For **double** data, linearly interpolates the value by using the two corresponding samples
- For **Boolean** data, uses **false** for the first half of the time between two time values and **true** for the second half
- For a built-in data type other than **double** or **Boolean**:
  - Upcasts the data to **double**
  - Performs linear interpolation (as described for **double** data)
• Downcasts the interpolated value to the original data type

You cannot use linear interpolation with enumerated (`enum`) data.

The block uses the value of the last known data point as the value of time hits that occur after the last known data point.

To determine the block output after the last time hit for which workspace data is available, combine the settings of these parameters:

• **Interpolate data**
• **Form output after final data value by**

For details, see the **Form output after final data value by** parameter.

**Programmatic Use**

Block Parameter: Interpolate
Type: character vector
Values: 'off' | 'on'
Default: 'on'

**Enable zero-crossing detection** — Enable zero-crossing detection

When you select **Enable zero-crossing detection**, and the input array contains multiple entries for the same time hit, Simulink detects a zero crossing. For example, suppose that the input array has this data.

<table>
<thead>
<tr>
<th>time</th>
<th>0 1 2 2 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>signal</td>
<td>2 3 4 5 6</td>
</tr>
</tbody>
</table>

At time 2, there is a zero crossing from input signal discontinuity. For more information, see “Zero-Crossing Detection”.

For bus signals, Simulink detects zero crossings across all leaf bus elements.

**Programmatic Use**

Block Parameter: ZeroCross
Type: character vector
Values: 'off' | 'on'
Default: 'on'
**Form output after final data value by** — Determine block output after final data point

Extrapolation (default) | Setting to zero | Holding final value | Cyclic repetition

The From Workspace block uses the **Interpolate data** and **Form output after final data value by** settings to determine the block output after the last sample available in the workspace data.

This table lists the block output, based on the values of the two options.

<table>
<thead>
<tr>
<th>Setting for Form Output After Final Data Value By</th>
<th>Setting for Interpolate Data</th>
<th>Block Output After Final Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extrapolation</td>
<td>On</td>
<td>Extrapolated from final data value</td>
</tr>
<tr>
<td></td>
<td>Off</td>
<td>Error</td>
</tr>
<tr>
<td>Setting to zero</td>
<td>On</td>
<td>Zero</td>
</tr>
<tr>
<td></td>
<td>Off</td>
<td></td>
</tr>
<tr>
<td>Holding final value</td>
<td>On</td>
<td>Final value from workspace</td>
</tr>
<tr>
<td></td>
<td>Off</td>
<td></td>
</tr>
<tr>
<td>Cyclic repetition</td>
<td>On</td>
<td>Error</td>
</tr>
<tr>
<td></td>
<td>Off</td>
<td>Repeated from workspace if the workspace data is in structure-without-time format. Error otherwise.</td>
</tr>
</tbody>
</table>

For example, when you enable **Interpolate data** and set **Form output after final data value by** to **Extrapolation**, the block uses the last two data points in the workspace data to extrapolate values for subsequent data points.

For example, consider this model. The From Workspace block reads workspace data that was created by logging the output of the Sine Wave block for a 16-second simulation. The Dashboard Scope blocks display the Sine Wave block output and From Workspace block output for a 20-second simulation. After the simulation reaches 16 seconds, the From Workspace block output diverges from the Sine Wave block output as the From Workspace block linearly extrapolates values for the remainder of the simulation.
**Programmatic Use**

**Block Parameter:** OutputAfterFinalValue  
**Type:** character vector  
**Values:** 'Extrapolation' | 'Setting to zero' | 'Holding final value' | 'Cyclic repetition'  
**Default:** 'Extrapolation'

**Block Characteristics**

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>bus</th>
<th>double</th>
<th>enumerated</th>
<th>fixed point&lt;sup&gt;a&lt;/sup&gt;</th>
<th>integer</th>
<th>single</th>
<th>string</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<sup>a</sup> Fixed point is supported only when the data type is fixed.
<table>
<thead>
<tr>
<th>Multidimensional Signals</th>
<th>yes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variable-Size Signals</td>
<td>yes</td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>yes</td>
</tr>
</tbody>
</table>

a. Supports input via `fi` objects created using Fixed-Point Designer.

### Extended Capabilities

#### Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

Supports input via `fi` objects created using Fixed-Point Designer.

### See Also

Environment Controller | From File | From Spreadsheet | To File | To Workspace

### Topics

“Overview of Signal Loading Techniques”
“Comparison of Signal Loading Techniques”
“Determine Where to Store Variables and Objects for Simulink Models”
“Use From Workspace Block for Test Case”
“Load Data Using the From Workspace Block”
“Load Signal Data That Uses Units”
“Load Signal Data for Simulation”

**Introduced before R2006a**
Function-Call Feedback Latch

Break feedback loop involving data signals between function-call blocks

Library: Simulink / Ports & Subsystems

Description

Use the Function-Call Feedback Latch block to break a feedback loop of data signals between one or more function-call blocks. Specifically, break a feedback loop formed in one of the following ways.

- **When function-call blocks connect to branches of the same function-call signal**

  Place the Function-Call Feedback Latch block on the feedback signal between the branched blocks. As a result, the latch block delays the signal at the input of the destination function-call block, and the destination function-call block executes prior to the source function-call block of the latch block.

To run this model, see “Function-Call Blocks Connected to Branches of the Same Function-Call Signal” on page 14-197.
- **When the loop involves parent and child function-call blocks, where the child initiator is inside the parent**

Place the Function-Call Feedback Latch block on the feedback signal between the child and the parent. This arrangement prevents the signal value, read by the parent (FCSS1), from changing during execution of the child. In other words, the parent reads the value from the previous execution of the child (FCSS2).

To run this model, see “Function-Call Feedback Latch on Feedback Signal Between Child and Parent” on page 14-198.

Using the latch block is equivalent to selecting the **Latch input for function-call feedback signals** check box on the Inport block in the destination function-call subsystem or model. However, an advantage of the latch block over using the dialog parameter is that one can design the destination function-call subsystem or model in a modular fashion and then use it either in or out of the context of loops.

The Function-Call Feedback Latch block is better suited than Unit Delay or Memory blocks in breaking function-call feedback loops for the following reasons:

- The latch block delays the feedback signal for exactly one execution of the source function-call block. This behavior is different from the Unit Delay or Memory blocks for cases where the function-call subsystem blocks may execute multiple times in a given simulation step.
• Unlike the Unit Delay or Memory blocks, the latch block may be used to break loops involving asynchronous function-call subsystems.
• The latch block can result in better performance, in terms of memory optimization, for generated code.

Ports

Input

_In — Signal from a function-call subsystem block_

scalar | vector | matrix

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus

Output

_Out — Signal to a function-call subsystem_

scalar | vector | matrix

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus

Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>bus</th>
<th>double</th>
<th>enumerated</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

See Also
Function-Call Feedback Latch | Function-Call Generator | Function-Call Split | Function-Call Subsystem | Trigger

Topics
“Using Function-Call Subsystems”

Introduced in R2011a
Function-Call Generator

Provide function-call events to control execution of a subsystem or model

**Library:** Simulink / Ports & Subsystems

### Description

The Function-Call Generator block provides function-call events that execute a function-call subsystem or function-call model at the rate that you specify with the **Sample time** parameter. To iteratively execute each function-call block multiple times at each time step, use the **Number of Iterations** parameter.

To execute multiple function-call subsystems or models in a specified order, use the Function-Call Generator block with a Function-Call Split block. For an example, see the Function-Call Split block documentation.

### Ports

**Output**

**Function Call** — Sends function-call events to the function-call input port of a function-call subsystem or function-call model

Scalar

### Parameters

**Sample time** — Specify time interval

-1 (default) | Ts | [Ts, To]

Specify the time interval between function call events to a subsystem or model containing a Trigger block with **Trigger type** set to function-call. If the actual calling rate for
the subsystem or model differs from the time interval this parameter specifies, Simulink displays an error.

**Settings**

-1

    Inherit time interval from the model.

Ts

    Scalar where Ts is the time interval.

[Ts, To]

    Vector where Ts is the time interval and To is the initial time offset.

**Programmatic Use**

**Block Parameter:** `sample_time`

**Type:** character vector

**Values:** `-1` | `'Ts'` | `[Ts, To]`

**Default:** `-1`

**Number of iterations — Specify number of times to provide a function-call at each time step**

1 (default) | integer

The value of this parameter can be a vector where each element of the vector specifies a number of times to execute a function-call subsystem. The total number of times that a function-call subsystem executes per time step is equal to the sum of vector element values.

Suppose that you specify the number of iterations to be `[2 2]` and connect the output of this block to the control port of a function-call subsystem. In this case, the function-call subsystem executes four times at each time step.

**Settings**

1

    Provide function-call once during each time step.

integer

    Signed or unsigned integer number. Provide the specified number of function calls at each time step.
Programmatic Use
Block Parameter: numberOfIterations
Type: character vector
Values: '1' | '<integer>'
Default: '1'

Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>no</td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
</tr>
</tbody>
</table>

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also
Function-Call Feedback Latch | Function-Call Generator | Function-Call Split | Function-Call Subsystem | Trigger

Topics
“Using Function-Call Subsystems”
Introduced before R2006a
Function-Call Split

Provide junction for splitting function-call line

Library: Simulink / Ports & Subsystems

Description

The Function-Call Split block allows a function-call line to branch and connect to several function-call subsystems or function-call models.

The function-call subsystem or function-call model connected to the output port of the Function-Call Split block that is marked with a dot execute before the subsystems or models connected to other output ports. If data dependencies between subsystems or models do not support the specified execution order, the Function-Call Split block returns an error. To eliminate this error, consider selecting the Latch input for feedback signals of function-call subsystem outputs parameter on one or more Inport blocks of the function-call subsystems models involved in a data-dependency loop. Selecting this option delays the corresponding function-call, thereby eliminating the data-dependency loop.

For a model to contain Function-Call Split blocks, you must set the following diagnostic to error: Model Configuration Parameters > Diagnostics > Connectivity > Invalid function-call connection.

If you select the model menu option Display > Blocks > Sorted Execution Order, then the execution order of function-call subsystems connected to branches of a given function-call signal appears on the blocks. Each subsystem has an execution order of the form s: [B#] where # is a number that ranges from 0 to one less than the total number of subsystems or models connected to branches of a given signal. The subsystems execute in ascending order based on this number.

The Function-Call Split block supports “Signal Label Propagation”.

The following model shows how to apply the Latch input for feedback signals of function-call subsystem outputs parameter to work around a data-dependency error caused by using a Function-Call Split block. By turning this parameter on in the f1 subsystem Inport block, the Function-Call Split block ignores the data dependency of
signal b. The block breaks the loop of data dependencies between subsystems f1 and g1. The model achieves the behavior of consistently calling f1 to execute before g1. For a given execution step, subsystem f1 uses the g1 output computed at the previous execution step.

Limitations

The Function-Call Split block has these limitations:

- All function-call subsystems and models connected to a given function-call signal must reside within the same nonvirtual layer of the model hierarchy.
- You cannot connect branched function-call subsystems or models and their children directly back to the function-call initiator.
- Function-call subsystems and models connected to branches of a function-call signal cannot have multiple (muxed) initiators.
- A Function-Call Split block cannot have its input from a signal with multiple function-call elements.
Ports

Input

Function Call — Function-call line
scalar

A Function-Call Generator block or a Stateflowchart can provide function-call events.

Output

Function Call — Function-call signal
scalar

Function-call line connected to a function-call subsystem or function-call model.

Parameters

Icon shape — Select block icon shape
distinctive (default) | round

Select block icon shape.

Settings
distinctive
   Rectangular block icon.
round
   Circular block icon.

Programmatic Use
Block Parameter: IconShape
Type: character vector
Values: 'distinctive' | 'round'
Default: 'distinctive'

Number of output ports — Specify number of output ports
2 (default) | integer
Specify number of function-call signal output ports.

**Settings**

2

Two function-call output ports.

**integer**

Integer number

**Programmatic Use**

**Block Parameter**: NumOutputPorts  
**Type**: character vector  
**Values**: '2' | '<integer>'  
**Default**: '2'

**Output port layout — Select order of output ports**

**default (default) | reverse**

Select the order of function-call output ports with respect to which port provides a function-call first.

**Settings**

**default**

Top port provides function-call first.

**reverse**

Bottom port provides function-call first.

**Programmatic Use**

**Block Parameter**: OutputPortLayout  
**Type**: character vector  
**Values**: 'default' | 'reverse'  
**Default**: 'default'

**Block Characteristics**

<table>
<thead>
<tr>
<th>Data Types</th>
<th>double</th>
</tr>
</thead>
</table>
Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also
Function-Call Feedback Latch | Function-Call Generator | Function-Call Subsystem

Topics
“Using Function-Call Subsystems”

Introduced in R2010a
Function-Call Subsystem

Subsystem whose execution is controlled by an external function-call input

Library: Simulink / Ports & Subsystems

Description

The Function-Call Subsystem block is a Subsystem block preconfigured as a starting point for creating a subsystem that executes when a function-call input port receives a function-call event. A Stateflow chart, Function-Call Generator block, or an S-function block can provide function-call events. See “Using Function-Call Subsystems”.

Use Function-Call Subsystem blocks to:

• Schedule the execution order of model components.
• Control the rate of model component execution.

Simulink ignores a priority set on a Function-Call Subsystem block. Instead, set the priority on a Inport block connected to the control port. The Inport block can receive a function-call event from a Stateflow® chart, Function-Call Generator block, or an S-function block.
For an explanation of Function-Call Subsystem block parameters, see Subsystem, Atomic Subsystem, Nonvirtual Subsystem, CodeReuse Subsystem.

Ports

Input

In — Signal input to a subsystem block
 scalar | vector | matrix

Placing an Inport block in a subsystem block adds an external input port to the block. The port label matches the name of the Inport block.

Use Inport blocks to get signals from the local environment.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 |
 Boolean | fixed point | enumerated | bus

Function Call — Function-call input port to a subsystem block
 scalar

Placing a Trigger block in a subsystem block adds an external input port to the block. Selecting function-call from the Trigger type list, changes the block to a Function-Call port block that accepts function-call events.

Output

Out — Signal output from a subsystem
 scalar | vector | matrix

Placing an Outport block in a subsystem block adds an output port from the block. The port label on the subsystem block is the name of the Outport block.

Use Outport blocks to send signals to the local environment.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 |
 Boolean | fixed point | enumerated | bus
Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean(^a)</th>
<th>bus(^a)</th>
<th>double(^a)</th>
<th>enumerated(^a)</th>
<th>fixed point(^a)</th>
<th>integer(^a)</th>
<th>single(^a)</th>
<th>string(^a)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>limited(^a)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>limited(^a)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\(^a\) Actual data type or capability support depends on block implementation.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

Actual code generation support depends on block implementation.

PLC Code Generation
Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

Actual data type support depends on block implementation.
See Also
Function-Call Feedback Latch | Function-Call Generator | Function-Call Split | Subsystem, Atomic Subsystem, Nonvirtual Subsystem, CodeReuse Subsystem

Topics
“Using Function-Call Subsystems”

Introduced before R2006a
Function Caller

Call Simulink or exported Stateflow function

Library: Simulink / User-Defined Functions

Description

A Function Caller block calls and executes a function defined with a Simulink Function block or an exported Stateflow function. Using Function Caller blocks, you can call a function from anywhere in a model or chart hierarchy.

Ports

Input

Input argument — Input signal for an input argument

scalar | vector | matrix

Input signal for an input argument that is sent to the function.

The function prototype determines the number and name of input ports that appear on the Function Caller block. Connect signal lines to the input ports to send data to a function through the function input arguments.

For example, \( y = \text{myfunction}(u) \) creates one input port \( (u) \) on the Function Caller block.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus
Output

**Output argument — Output signal for an output argument**

scalar | vector | matrix

Output signal for an output argument that the function returns.

The function prototype determines the number and name of output ports that appear on the Function Caller block. Connect signal lines to the output ports to receive data from a function through the function output arguments.

For example, \( y = \text{myfunction}(u) \) creates one output port \( y \) on the Function Caller block.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `Boolean` | `fixed point` | `enumerated` | `bus`

Parameters

**Function prototype — Specify function interface**

\( y=f(u) \) (default) | <function prototype>

Specify the function interface between a Function Caller block and a Simulink function. A Simulink function can be a Simulink Function block, an exported Stateflow graphical function, or an exported Stateflow MATLAB function. For a call to a Simulink Function block:

- Function call argument names must match the function arguments.
- Function names, input arguments, and output arguments must be valid MATLAB identifiers.

**Programmatic Use**

**Block Parameter:** `FunctionPrototype`

**Type:** character vector

**Values:** `'y=f(u)'` | `<function prototype>`

**Default:** `'y=f(u)'`

**Input argument specifications — Specify input argument data type, dimensions, and complexity**

<Enter example> (default) | <MATLAB expression>
Specify a comma-separated list of MATLAB expressions that combine data type, dimensions, and complexity (real or imaginary) for each input argument. For examples, see “Argument Specification for Simulink Function Blocks”.

This specification must match the Simulink Function block data type specified with the **Data type** parameter of the Argument Inport block.

**Programmatic Use**

**Block Parameter:** 'InputArgumentSpecifications'

**Type:** character vector

**Values:** ' ' | '<MATLAB expression>'

**Default:** ' '

**Output argument specifications — Specify output argument data type, dimensions, and complexity**

<Enter example> (default) | <MATLAB Expression>

Specify a comma-separated list of MATLAB expressions that combine data type, dimensions, and complexity (real or imaginary) for each output argument. For examples, see “Argument Specification for Simulink Function Blocks”.

This specification must match the Simulink Function block data type specified with the **Data type** parameter of the Argument Outport Block.

**Programmatic Use**

**Block Parameter:** 'OutputArgumentSpecifications'

**Type:** character vector

**Values:** ' ' | '<MATLAB expression>'

**Default:** ' '

**Sample time — Time interval between function calls**

-1 (default) | Ts | [Ts, To]

Specify the time interval between function calls to a subsystem or model containing this Trigger block. If the actual calling rate for the subsystem or model differs from the time interval this parameter specifies, Simulink displays an error.

**Settings**

-1

Inherit time interval from the trigger signal.
Ts
Scalar where Ts is the time interval.

[Ts, To]
Vector where Ts is the time interval and To is the initial time offset.

**Programmatic Use**

**Block Parameter:** SampleTime
**Type:** character vector
**Values:** '-1' | 'Ts' | '[Ts, To]' 
**Default:** '-1'

**Block Characteristics**

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>bus</th>
<th>double</th>
<th>enumerated</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Extended Capabilities**

**C/C++ Code Generation**
Generate C and C++ code using Simulink® Coder™.

**Fixed-Point Conversion**
Design and simulate fixed-point systems using Fixed-Point Designer™.
See Also
Argument Inport | Argument Outport | Chart | Function Caller | Function-Call Subsystem | Inport | MATLAB Function | Outport | Subsystem | Trigger

Topics
“Simulink Functions Overview”
“Scoped Simulink Function Blocks in Models”
“Simulink functions: Simulink Function block, exported Stateflow graphical and MATLAB functions”
Gain

Multiply input by constant

Library: Simulink / Commonly Used Blocks
Simulink / Math Operations
HDL Coder / Commonly Used Blocks
HDL Coder / HDL Floating Point Operations
HDL Coder / Math Operations

Description

The Gain block multiplies the input by a constant value (gain). The input and the gain can each be a scalar, vector, or matrix.

You specify the value of gain in the Gain parameter. The Multiplication parameter lets you specify element-wise or matrix multiplication. For matrix multiplication, this parameter also lets you indicate the order of the multiplicands.

Gain is converted from doubles to the data type specified in the block mask offline using round-to-nearest and saturation. The input and gain are then multiplied, and the result is converted to the output data type using the specified rounding and overflow modes.

Ports

Input

Port_1 — Input signal
scalar | vector | matrix

The Gain block accepts real or complex-valued scalar, vector, or matrix input. The Gain block supports fixed-point data types. If the input of the Gain block is real and gain is complex, the output is complex.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point
Output

Port_1 — Input multiplied by gain
scalar | vector | matrix

The Gain block outputs the input multiplied by a constant gain value. When the input to the Gain block is real and gain is complex, the output is complex.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point

Parameters

Main

Gain — Value by which to multiply the input
1 (default) | real or complex-valued scalar, vector, or matrix

Specify the value by which to multiply the input. Gain can be a real or complex-valued scalar, vector, or matrix.

Programmatic Use

Block Parameter: Gain
Type: character vector
Values: '1' | real- or complex-valued scalar, vector, or matrix
Default: '1'

Multiplication — Specify the multiplication mode
Element-wise(K.*u) (default) | Matrix(K*u) | Matrix(u*K) | Matrix(K*u) (u vector)

Specify one of these multiplication modes:

- Element-wise(K.*u) — Each element of the input is multiplied by each element of the gain. The block performs expansions, if necessary, so that the input and gain have the same dimensions.
- Matrix(K*u) — The input and gain are matrix-multiplied with the input as the second operand.
- Matrix(u*K) — The input and gain are matrix-multiplied with the input as the first operand.
- **Matrix(K*u) (u vector)** — The input and gain are matrix multiplied with the input as the second operand. This mode is identical to Matrix(K*u), except for how dimensions are determined.

Suppose that K is an $m \times n$ matrix. Matrix(K*u)(u vector) sets the input to a vector of length $n$ and the output to a vector of length $m$. In contrast, Matrix(K*u) uses propagation to determine dimensions for the input and output. For an $m \times n$ gain matrix, the input can propagate to an $n \times q$ matrix, and the output becomes an $m \times q$ matrix.

**Programmatic Use**

**Parameter:** Multiplication  
**Type:** character vector  
**Value:** 'Element-wise(K.*u)' | 'Matrix(K*u)' | 'Matrix(u*K)' | 'Matrix(K*u) (u vector)'  
**Default:** 'Element-wise(K.*u)'

**Sample time — Specify sample time as a value other than -1**  
-1 (default) | scalar | vector

Specify the sample time as a value other than -1. For more information, see “Specify Sample Time”.

**Dependencies**

This parameter is not visible unless it is explicitly set to a value other than -1. To learn more, see “Blocks for Which Sample Time Is Not Recommended”.

**Programmatic Use**

**Block Parameter:** SampleTime  
**Type:** character vector  
**Values:** scalar or vector  
**Default:** '-1'

**Signal Attributes**

**Output minimum — Minimum output value for range checking**  
[] (default) | scalar

Lower value of the output range that Simulink checks.

Simulink uses the minimum to perform:
- Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters”) for some blocks.
- Simulation range checking (see “Signal Ranges” and “Enable Simulation Range Checking”).
- Automatic scaling of fixed-point data types.
- Optimization of the code that you generate from the model. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. For more information, see “Optimize using the specified minimum and maximum values” (Simulink Coder).

**Note** Output minimum does not saturate or clip the actual output signal. Use the Saturation block instead.

**Programmatic Use**

**Block Parameter:** OutMin  
**Type:** character vector  
**Values:** 
- `[' ']'` scalar  
- Default: `[' ']`

**Output maximum — Maximum output value for range checking**

[] (default) | scalar

Upper value of the output range that Simulink checks.

Simulink uses the maximum value to perform:
- Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters”) for some blocks.
- Simulation range checking (see “Signal Ranges” and “Enable Simulation Range Checking”).
- Automatic scaling of fixed-point data types.
- Optimization of the code that you generate from the model. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. For more information, see “Optimize using the specified minimum and maximum values” (Simulink Coder).
**Note** Output maximum does not saturate or clip the actual output signal. Use the Saturation block instead.

**Programmatic Use**

**Block Parameter:** OutMax  
**Type:** character vector  
**Values:** '[ ]' | scalar  
**Default:** '[ ]'

**Output data type — Specify the output data type**

Inherit: Inherit via internal rule (default) | Inherit: Keep MSB | Inherit: Match scaling | Inherit: Inherit via back propagation | Inherit: Same as input | double | single | int8 | uint8 | int16 | uint16 | int32 | int64 | uint64 | uint32 | fixdt(1,16) | fixdt(1,16,0) | fixdt(1,16,2^0,0) | <data type expression>

Choose the data type for the output. The type can be inherited, specified directly, or expressed as a data type object such as Simulink.NumericType.

When you select an inherited option, the block exhibits these behaviors:

- **Inherit: Inherit via internal rule** — Simulink chooses a data type to balance numerical accuracy, performance, and generated code size, while taking into account the properties of the embedded target hardware. If you change the embedded target settings, the data type selected by the internal rule might change. For example, if the block multiplies an input of type int8 by a gain of int16 and ASIC/FPGA is specified as the targeted hardware type, the output data type is sfix24. If Unspecified (assume 32-bit Generic), in other words, a generic 32-bit microprocessor, is specified as the target hardware, the output data type is int32. If none of the word lengths provided by the target microprocessor can accommodate the output range, Simulink software displays an error in the Diagnostic Viewer.

- **Inherit: Keep MSB**— Simulink chooses a data type that maintains the full range of the operation, then reduces the precision of the output to a size appropriate for the embedded target hardware.

**Tip** For more efficient generated code, deselect the Saturate on integer overflow parameter.

This rule never produces overflows.
• **Inherit: Match scaling**— Simulink chooses a data type whose scaling matches the scaling of the input types. If the full range of the type does not fit on the embedded target hardware, the range is reduced yielding a type appropriate for the embedded target hardware. This rule can produce overflows.

It is not always possible for the software to optimize code efficiency and numerical accuracy at the same time. If these internal rules do not meet your specific needs for numerical accuracy or performance, use one of the following options:

• Specify the output data type explicitly.
• Use the simple choice of **Inherit: Same as input**.
• Explicitly specify a default data type such as `fixdt(1,32,16)` and then use the Fixed-Point Tool to propose data types for your model. For more information, see `fxptdlg`.
• To specify your own inheritance rule, use **Inherit: Inherit via back propagation** and then use a Data Type Propagation block. Examples of how to use this block are available in the Signal Attributes library Data Type Propagation Examples block.
• **Inherit: Inherit via back propagation** — Use data type of the driving block.
• **Inherit: Same as input** — Use data type of input signal.

**Programmatic Use**

**Block Parameter**: OutDataTypeStr

**Type**: character vector

**Values**: 'Inherit: Inherit via internal rule' | 'Inherit: Keep MSB' | 'Inherit: Match scaling' | 'Inherit: Same as input' | 'Inherit: Inherit via back propagation' | 'single' | 'int8' | 'uint8' | 'int16' | 'uint16' | 'int32' | 'uint32' | 'int64' | 'uint64' | 'fixdt(1,16)' | 'fixdt(1,16,0)' | 'fixdt(1,16,2^0,0)' | '<data type expression>'

**Default**: 'Inherit: Inherit via internal rule'

Lock output data type setting against changes by the fixed-point tools — Prevent fixed-point tools from overriding Output data type

off (default) | on

Select this parameter to prevent the fixed-point tools from overriding the **Output** data type you specify on the block. For more information, see “Use Lock Output Data Type Setting” (Fixed-Point Designer).
Programmatic Use

Block Parameter: LockScale
Type: character vector
Values: 'off' | 'on'
Default: 'off'

Integer rounding mode — Rounding mode for fixed-point operations
Floor (default) | Ceiling | Convergent | Nearest | Round | Simplest | Zero

Specify the rounding mode for fixed-point operations. For more information, see “Rounding” (Fixed-Point Designer).

Block parameters always round to the nearest representable value. To control the rounding of a block parameter, enter an expression using a MATLAB rounding function into the mask field.

Programmatic Use

Block Parameter: RndMeth
Type: character vector
Values: 'Ceiling' | 'Convergent' | 'Floor' | 'Nearest' | 'Round' | 'Simplest' | 'Zero'
Default: 'Floor'

Saturate on integer overflow — Method of overflow action
off (default) | on

Specify whether overflows saturate or wrap.
<table>
<thead>
<tr>
<th>Action</th>
<th>Rationale</th>
<th>Impact on Overflows</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Select this check box (on).</td>
<td>Your model has possible overflow, and you want explicit saturation protection in the generated code.</td>
<td>Overflows saturate to either the minimum or maximum value that the data type can represent.</td>
<td>The maximum value that the int8 (signed, 8-bit integer) data type can represent is 127. Any block operation result greater than this maximum value causes overflow of the 8-bit integer. With the check box selected, the block output saturates at 127. Similarly, the block output saturates at a minimum output value of -128.</td>
</tr>
<tr>
<td>Do not select this check box (off).</td>
<td>You want to optimize efficiency of your generated code. You want to avoid overspecifying how a block handles out-of-range signals. For more information, see “Check for Signal Range Errors”.</td>
<td>Overflows wrap to the appropriate value that is representable by the data type.</td>
<td>The maximum value that the int8 (signed, 8-bit integer) data type can represent is 127. Any block operation result greater than this maximum value causes overflow of the 8-bit integer. With the check box cleared, the software interprets the overflow-causing value as int8, which can produce an unintended result. For example, a block result of 130 (binary 1000 0010) expressed as int8, is -126.</td>
</tr>
</tbody>
</table>

When you select this check box, saturation applies to every internal operation on the block, not just the output, or result. Usually, the code generation process can detect when overflow is not possible. In this case, the code generator does not produce saturation code.
Programmatic Use

Block Parameter: SaturateOnIntegerOverflow
Type: character vector
Values: 'off' | 'on'
Default: 'off'

Parameter Attributes

Parameter minimum — Specify the minimum value of gain
[] (default) | scalar

Specify the minimum value of gain. The default value is [] (unspecified). Simulink uses this value to perform:

- Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters”)
- Automatic scaling of fixed-point data types

Programmatic Use

Block Parameter: ParamMin
Type: character vector
Value: scalar
Default: '['

Parameter maximum — Specify the maximum value of gain
[] (default) | scalar

Specify the maximum value of gain. The default value is [] (unspecified). Simulink uses this value to perform:

- Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters”)
- Automatic scaling of fixed-point data types

Programmatic Use

Block Parameter: ParamMax
Type: character vector
Value: scalar
Default: '['
Parameter data type — Specify the data type of the Gain parameter

Inherit: Inherit via internal rule (default) | Inherit: Same as input | Inherit: Inherit from 'Gain' | double | single | int8 | uint8 | int16 | uint16 | int32 | uint32 | int64 | uint64 | fixdt(1,16) | fixdt(1,16,0) | fixdt(1,16,2^0,0) | <data type expression>

Specify the data type of the Gain parameter.

Programmatic Use

Block Parameter: ParamDataTypeStr
Type: character vector
Values: 'Inherit: Inherit via internal rule' | 'Inherit: Same as input' | 'Inherit: Inherit via back propagation' | 'single' | 'int8' | 'uint8' | 'int16' | 'uint16' | 'int32' | 'uint32' | 'int64' | 'uint64' | 'fixdt(1,16)' | 'fixdt(1,16,0)' | 'fixdt(1,16,2^0,0)' | <data type expression>
Default: 'Inherit: Inherit via internal rule'

Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>double</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.
**HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

**Tunable Parameters**

You can use a tunable parameter in a Gain block intended for HDL code generation. For details, see “Generate DUT Ports for Tunable Parameters” (HDL Coder).

**HDL Architecture**

<table>
<thead>
<tr>
<th>ConstMultiplierOptimization</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>none (Default)</td>
<td>By default, HDL Coder does not perform CSD or FCSD optimizations. Code generated for the Gain block retains multiplier operations.</td>
</tr>
<tr>
<td>csd</td>
<td>When you specify this option, the generated code decreases the area used by the model while maintaining or increasing clock speed, using canonical signed digit (CSD) techniques. CSD replaces multiplier operations with add and subtract operations. CSD minimizes the number of addition operations required for constant multiplication by representing binary numbers with a minimum count of nonzero digits.</td>
</tr>
<tr>
<td>fcsd</td>
<td>This option uses factored CSD (FCSD) techniques, which replace multiplier operations with shift and add/subtract operations on certain factors of the operands. These factors are generally prime but can also be a number close to a power of 2, which favors area reduction. You can achieve a greater area reduction with FCSD at the cost of decreasing clock speed.</td>
</tr>
</tbody>
</table>
When you specify this option, the coder chooses between the CSD or FCSD optimizations. The coder chooses the optimization that yields the most area-efficient implementation, based on the number of adders required. When you specify **auto**, the coder does not use multipliers, unless conditions are such that CSD or FCSD optimizations are not possible (for example, if the design uses floating-point arithmetic).

### HDL Block Properties

<table>
<thead>
<tr>
<th><strong>ConstMultiplierOptimization</strong></th>
<th><strong>Description</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>auto</strong></td>
<td>When you specify this option, the coder chooses between the CSD or FCSD optimizations. The coder chooses the optimization that yields the most area-efficient implementation, based on the number of adders required. When you specify <strong>auto</strong>, the coder does not use multipliers, unless conditions are such that CSD or FCSD optimizations are not possible (for example, if the design uses floating-point arithmetic).</td>
</tr>
</tbody>
</table>

### General

<table>
<thead>
<tr>
<th><strong>ConstMultiplierOptimization</strong></th>
<th>Canonical signed digit (CSD) or factored CSD optimization. The default is none. See also “ConstMultiplierOptimization” (HDL Coder).</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ConstrainedOutputPipeline</strong></td>
<td>Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. For more details, see “ConstrainedOutputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td><strong>DSPStyle</strong></td>
<td>Synthesis attributes for multiplier mapping. The default is none. See also “DSPStyle” (HDL Coder).</td>
</tr>
<tr>
<td><strong>InputPipeline</strong></td>
<td>Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “InputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td><strong>OutputPipeline</strong></td>
<td>Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “OutputPipeline” (HDL Coder).</td>
</tr>
</tbody>
</table>

**Note** For certain values of the **Gain** parameter, native floating point implements the algorithm differently instead of using multipliers. For example, if you set the **Gain** parameter to 1, the generated model uses a wire to pass the input to the output. If you set the **Gain** parameter to -1, the generated model shows a Unary Minus block that inverts the polarity of the input signal. This implementation reduces the latency and resource usage on the target platform.
Native Floating Point

**HandleDenormals**
Specify whether you want HDL Coder to insert additional logic to handle denormal numbers in your design. Denormal numbers are numbers that have magnitudes less than the smallest floating-point number that can be represented without leading zeros in the mantissa. The default is inherit. See also “HandleDenormals” (HDL Coder).

**LatencyStrategy**
Specify whether to map the blocks in your design to inherit, Max, Min, Zero, or Custom for the floating-point operator. The default is inherit. See also “LatencyStrategy” (HDL Coder).

**NFPCustomLatency**
To specify a value, set LatencyStrategy to Custom. HDL Coder adds latency equal to the value that you specify for the NFPCustomLatency setting. See also “NFPCustomLatency” (HDL Coder).

**MantissaMultiplyStrategy**
Specify how to implement the mantissa multiplication operation during code generation. By using different settings, you can control the DSP usage on the target FPGA device. The default is inherit. See also “MantissaMultiplyStrategy” (HDL Coder).

Complex Data Support
This block supports code generation for complex signals.

PLC Code Generation
Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

See Also
Product | Slider Gain

Topics
“Model a Continuous System”
Introduced before R2006a
Gauge

Display input value on circular scale

Library: Simulink / Dashboard

Description

The Gauge block displays the connected signal on a circular scale during simulation. You can use the Gauge block with other Dashboard blocks to build an interactive dashboard of controls and indicators for your model. The Gauge block provides an indication of the instantaneous value of the connected signal throughout simulation. You can modify the range of the Gauge block to fit your data. You can also change the appearance of the dial to provide more information about your signal. For example, you can color-code in-specification and out-of-specification ranges.

Connecting Dashboard Blocks

Dashboard blocks do not use ports to connect to signals. To connect Dashboard blocks to signals in your model, use connect mode. Connect mode facilitates the process of connecting Dashboard blocks in your model, especially when you want to connect multiple blocks at once. If you only want to connect a single Dashboard block, you can also use the Connection table in the block dialog box to make the connection.

Tip To facilitate understanding and debugging your model, you can modify Dashboard block connections in your model during normal and accelerator mode simulations.

To enter connect mode, click the Connect button that appears above your unconnected Dashboard block when you pause on it.
In connect mode, when you select one or more signals or blocks, a list of parameters and signals available for connection appears. Select a signal or parameter from the list to connect the selected Dashboard block. To connect another Dashboard block, pause on the block and click the **Connect** button above it. Then, make a selection of signals and blocks in your model, and choose a signal or parameter to connect.
When you finish connecting the Dashboard blocks in your model, click the Exit button in the upper-right of the canvas to exit connect mode.

Tip You can hide the message shown on unconnected blocks using the `set_param` function and the `ShowInitialText` block parameter.

Limitations

- Dashboard blocks can only connect to real scalar signals.
- You cannot use the `Connection` table to connect a Dashboard block to a block that is commented out. When you connect a Dashboard block to a commented block using connect mode, the Dashboard block does not display the connected value until you uncomment the block.
- Dashboard blocks cannot connect to signals inside referenced models.
- When you simulate a model hierarchy, Dashboard blocks inside referenced models do not update.
- Dashboard blocks do not support rapid accelerator simulation.
• Some signals are not available during accelerator mode simulations. To view such signals on a Dashboard block, mark the signal for logging.

**Parameters**

**Connection — Select a signal to connect and display**

Select the signal to connect using the **Connection** table. Populate the **Connection** table by selecting signals of interest in your model. Select the radio button next to the signal you want to display. Click **Apply** to connect the signal. To facilitate understanding and debugging your model, you can connect Dashboard blocks to signals in your model during simulation.

**Programmatic Use**

**Block Parameter:** Binding  
**Type:** Simulink.HMI.SignalSpecification  
**Default:** []

**Minimum — Minimum tick mark value**

0 (default) | scalar

Finite, real, double, scalar value specifying the minimum tick mark value for the arc. The minimum must be less than the value entered for the maximum.

**Programmatic Use**

To programmatically set the **Minimum** parameter, use a 1-by-3 vector containing values for the **Minimum**, **Tick Interval**, and **Maximum** parameters, in that order. To use the auto value for the **Tick Interval**, leave the **Tick Interval** position in the vector empty or specify -1.

**Block Parameter:** Limits  
**Type:** 1x3 vector  
**Default:** [0 -1 100]

**Maximum — Maximum tick mark value**

100 (default) | scalar

Finite, real, double, scalar value specifying the maximum tick mark value for the arc. The maximum must be greater than the value entered for the minimum.
**Programmatic Use**

To programmatically set the Maximum parameter, use a 1-by-3 vector containing values for the Minimum, Tick Interval, and Maximum parameters, in that order. To use the auto value for the Tick Interval, leave the Tick Interval position in the vector empty or specify -1.

**Block Parameter: Limits**
- **Type:** 1x3 vector
- **Default:** [0  -1 100]

**Tick Interval — Interval between major tick marks**
- **auto (default) | scalar**

Finite, real, positive, integer, scalar value specifying the interval of major tick marks on the arc. When set to auto, the block automatically adjusts the tick interval based on the minimum and maximum values.

**Programmatic Use**

To programmatically set the Tick Interval parameter, use a 1-by-3 vector containing values for the Minimum, Tick Interval, and Maximum parameters, in that order. To use the auto value for the Tick Interval, leave the Tick Interval position in the vector empty or specify -1.

**Block Parameter: Limits**
- **Type:** 1x3 vector
- **Default:** [0  -1 100]

**Scale Colors — Color indications on gauge arc**
- **colors for arc ranges**

Color specifications for ranges on the arc. Press the + button to add a color. For each color added, specify the minimum and maximum values of the range where you want to display that color.

**Programmatic Use**

To programmatically specify the Scale Colors parameter, use an array of structures with fields:

- **Min** — Minimum value for the color range on the scale.
- **Max** — Maximum value for the color range on the scale.
- **Color** — 1-by-3 vector of double values between 0 and 1 to specify the color for the range of the scale in the form [r g b]
Include a structure in the array for each scale range for which you want to specify a color.

```plaintext
rang1.Min = 0;
rang1.Max = 10;
rang1.Color = [0 0 1];
rang2.Min = 10;
rang2.Max = 15;
rang2.Color = [0 1 0];
scaleRanges = [rang1 range2];
```

**Block Parameter:** ScaleColors  
**Type:** structure array  
**Default:** 0x1 struct array

**Label — Block label position**  
Top (default) | Bottom | Hide

Position of the block label. When the block is connected to an element in the model, the label is the name of the connected element.

**Programmatic Use**  
**Block Parameter:** LabelPosition  
**Type:** character vector  
**Values:** 'Top' | 'Bottom' | 'Hide'  
**Default:** 'Top'

**Block Characteristics**

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>double</th>
<th>enumerated</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.
Dashboard blocks are ignored for code generation.

See Also
Custom Gauge | Half Gauge | Linear Gauge | Quarter Gauge

Topics
“Tune and Visualize Your Model with Dashboard Blocks”
“Decide How to Visualize Simulation Data”

Introduced in R2015a
Goto

Pass block input to From blocks

Library: Simulink / Signal Routing
        HDL Coder / Signal Routing

Description

The Goto block passes its input to its corresponding From blocks. The input can be a real-
or complex-valued signal or vector of any data type. From and Goto blocks allow you to
pass a signal from one block to another without actually connecting them.

A Goto block can pass its input signal to more than one From block, although a From
block can receive a signal from only one Goto block. The input to that Goto block is
passed to the From blocks associated with it as though the blocks were physically
connected. Goto blocks and From blocks are matched by the use of Goto tags.

The Tag Visibility parameter determines whether the location of From blocks that access
the signal is limited:

- **local** (default) — From and Goto blocks that use the same tag must be in the same
  subsystem. A local tag name is enclosed in brackets ([ ]).

- **scoped** — From and Goto blocks that use the same tag must be in the same
  subsystem or at any level in the model hierarchy below the Goto Tag Visibility block
  that does not entail crossing a nonvirtual subsystem boundary, in other words, the
  boundary of an atomic, conditionally executed, or function-call subsystem or a model
  reference. A scoped tag name is enclosed in braces ({}).

- **global** — From and Goto blocks using the same tag can be anywhere in the model
  except in locations that span nonvirtual subsystem boundaries.

The rule that From-Goto block connections cannot cross nonvirtual subsystem boundaries
has the following exception. A Goto block connected to a state port in one conditionally
executed subsystem is visible to a From block inside another conditionally executed
subsystem.
**Note** A scoped Goto block in a masked system is visible only in that subsystem and in the nonvirtual subsystems it contains. Simulink generates an error if you run or update a diagram that has a Goto Tag Visibility block at a higher level in the block diagram than the corresponding scoped Goto block in the masked subsystem.

Use local tags when the Goto and From blocks using the same tag name reside in the same subsystem. You must use global or scoped tags when the Goto and From blocks using the same tag name reside in different subsystems. When you define a tag as global, all uses of that tag access the same signal. A tag defined as scoped can be used in more than one place in the model.

The Goto block supports signal label propagation.

**Ports**

**Input**

**Port_1 — Input signal**

scalar | vector | matrix | N-D array

Input signal to be passed to the corresponding From block, specified as a scalar, vector, matrix, or N-D array.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus

**Parameters**

**Goto tag — Block identifier**

A (default) | ...

The Goto block identifier. This parameter identifies the Goto block whose scope is defined in this block.

**Programmatic Use**

**Block Parameter:** GotoTag

**Type:** character vector
Values: 'A' | ...  
Default: 'A'

**Tag visibility — Scope of the Goto block tag**

local (default) | scoped | global

The scope of the Goto block tag, specified as local, scoped, or global. When you set this parameter to scoped, you must use a Goto Tag Visibility block to define the scope of tag visibility.

**Programmatic Use**

Block Parameter: TagVisibility  
Type: character vector  
Values: 'local' | 'scoped' | 'global'  
Default: 'local'

**Icon display — Text to display on block icon**

Tag (default) | Signal name | Tag and signal name

Specifies the text to display on the block icon. The options are the block tag, the name of the signal that the block represents, or both the tag and the signal name.

**Programmatic Use**

Block Parameter: IconDisplay  
Type: character vector  
Values: 'Signal name' | 'Tag' | 'Tag and signal name'

Default: 'Tag'

**Rename All — Propagate name throughout model**

button

Rename the Goto tag. The new name propagates to all From and Goto Tag Visibility blocks that are listed in the Corresponding blocks box.

**Corresponding blocks — Blocks connected to this Goto block**

block path | ...

List of the From blocks and Goto Tag Visibility blocks connected to this Goto block. Click an entry in the list to display and highlight the corresponding From or Goto Tag Visibility block.
Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>bus</th>
<th>double</th>
<th>enumerated</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
<th>string</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture
This block has a single, default HDL architecture.
HDL Block Properties

| InputPipeline | Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “InputPipeline” (HDL Coder). |

PLC Code Generation
Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

See Also
From | Goto Tag Visibility

Topics
“Signal Label Propagation”

Introduced before R2006a
**Goto Tag Visibility**

Define scope of Goto block tag

**Library:** Simulink / Signal Routing

**Description**

The Goto Tag Visibility block defines the accessibility of Goto block tags that have **scoped** visibility. The value you specify for the **Goto tag** block parameter is accessible by From blocks in the same subsystem that contains the Goto Tag Visibility block and in subsystems below it in the model hierarchy.

A Goto Tag Visibility block is required for Goto blocks whose **Tag Visibility** parameter value is **scoped**. No Goto Tag Visibility block is needed if the tag visibility is either **local** or **global**. The block shows the tag name enclosed in braces ({}).

**Note** A **scoped** Goto block in a masked system is visible only in that subsystem and in the nonvirtual subsystems it contains. Simulink generates an error if you run or update a diagram that has a Goto Tag Visibility block at a lower level in the block diagram than the corresponding **scoped** Goto block in the masked subsystem.

**Parameters**

**Goto tag** — Goto block tag whose visibility is defined by the location of this block

A (default) | ...

The Goto block tag whose visibility is defined by the location of this block. From and Goto blocks using the specified tag must be in the same subsystem or at any level in the model hierarchy below the Goto Tag Visibility block that does not entail crossing a nonvirtual subsystem boundary, in other words, the boundary of an atomic, conditionally executed,
or function-call subsystem or a model reference. A scoped tag name is enclosed in braces ({}).

**Tip** If you use multiple From and Goto Tag Visibility blocks to refer to the same Goto tag, you can simultaneously rename the tag in all of the blocks. Use the **Rename All** button in the Goto block dialog box.

### Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>bus</th>
<th>double</th>
<th>enumerated</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Direct Feedthrough</strong></td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Multidimensional Signals</strong></td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Variable-Size Signals</strong></td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Zero-Crossing Detection</strong></td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Extended Capabilities

#### C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

#### PLC Code Generation
Generate Structured Text code using Simulink® PLC Coder™.

#### Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.
See Also
From | Goto

Topics
“Signal Label Propagation”

Introduced before R2006a
Ground

Ground unconnected input port

Library: Simulink / Commonly Used Blocks
        Simulink / Sources
        HDL Coder / Sources

Description

The Ground block connects to blocks whose input ports do not connect to other blocks. If you run a simulation with blocks that have unconnected input ports, Simulink issues warnings. Using a Ground block to ground those unconnected blocks can prevent these warnings.

Working with Fixed-Point Data Types

When working with fixed-point data types, there may be instances where the fixed-point data type cannot represent zero exactly. In these cases, the Ground block outputs a nonzero value that is the closest possible value to zero. This behavior applies only to fixed-point data types with nonzero bias. These expressions are examples of fixed-point data types that cannot represent zero:

- \texttt{fixd}(0, 8, 1, 1) — an unsigned 8-bit type with slope of 1 and bias of 1
- \texttt{fixd}(1, 8, 6, 3) — a signed 8-bit type with slope of 6 and bias of 3

Working with Enumerated Data Types

When working with enumerated data types, the Ground block outputs the default value of the enumeration. This behavior applies whether:

- The enumeration can represent zero
- The default value of the enumeration is zero

If the enumerated type does not have a default value, the Ground block outputs the first enumeration value in the type definition.
Ports

Output

Port_1 — Ground signal
scalar

The Ground block outputs a scalar signal with zero value, and the same data type as the port to which it connects.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus

Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>bus</th>
<th>double</th>
<th>enumerated</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
<th>string</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.
HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture
This block has a single, default HDL architecture.

HDL Block Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConstrainedOutput Pipeline</td>
<td>Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. For more details, see “ConstrainedOutputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td>OutputPipeline</td>
<td>Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “OutputPipeline” (HDL Coder).</td>
</tr>
</tbody>
</table>

Complex Data Support
This block supports code generation for complex signals.

PLC Code Generation
Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

See Also

Topics
“Unconnected block input ports”
Introduced before R2006a
Half Gauge

Display input value on semicircular scale

**Library:** Simulink / Dashboard

**Description**

The Half Gauge block displays the connected signal on a semicircular scale during simulation. You can use the Half Gauge block with other Dashboard blocks to build an interactive dashboard of controls and indicators for your model. The Half Gauge block provides an indication of the instantaneous value of the connected signal throughout simulation. You can modify the range of the Half Gauge block to fit your data. You can also customize the appearance of the Half Gauge block to provide more information about your signal. For example, you can color-code in-specification and out-of-specification ranges.

**Connecting Dashboard Blocks**

Dashboard blocks do not use ports to connect to signals. To connect Dashboard blocks to signals in your model, use connect mode. Connect mode facilitates the process of connecting Dashboard blocks in your model, especially when you want to connect multiple blocks at once. If you only want to connect a single Dashboard block, you can also use the **Connection** table in the block dialog box to make the connection.

**Tip** To facilitate understanding and debugging your model, you can modify Dashboard block connections in your model during normal and accelerator mode simulations.

To enter connect mode, click the **Connect** button that appears above your unconnected Dashboard block when you pause on it.
In connect mode, when you select one or more signals or blocks, a list of parameters and signals available for connection appears. Select a signal or parameter from the list to connect the selected Dashboard block. To connect another Dashboard block, pause on the block and click the Connect button above it. Then, make a selection of signals and blocks in your model, and choose a signal or parameter to connect.
When you finish connecting the Dashboard blocks in your model, click the **Exit** button in the upper-right of the canvas to exit connect mode.

**Tip** You can hide the message shown on unconnected blocks using the `set_param` function and the `ShowInitialText` block parameter.

### Limitations

- Dashboard blocks can only connect to real scalar signals.
- You cannot use the **Connection** table to connect a Dashboard block to a block that is commented out. When you connect a Dashboard block to a commented block using connect mode, the Dashboard block does not display the connected value until the you uncomment the block.
- Dashboard blocks cannot connect to signals inside referenced models.
- When you simulate a model hierarchy, Dashboard blocks inside referenced models do not update.
- Dashboard blocks do not support rapid accelerator simulation.
• Some signals are not available during accelerator mode simulations. To view such signals on a Dashboard block, mark the signal for logging.

Parameters

Connection — Select a signal to connect and display
signal connection options

Select the signal to connect using the Connection table. Populate the Connection table by selecting signals of interest in your model. Select the radio button next to the signal you want to display. Click Apply to connect the signal. To facilitate understanding and debugging your model, you can connect Dashboard blocks to signals in your model during simulation.

Programmatic Use
Block Parameter: Binding
Type: Simulink.HMI.SignalSpecification
Default: []

Minimum — Minimum tick mark value
0 (default) | scalar

Finite, real, double, scalar value specifying the minimum tick mark value for the arc. The minimum must be less than the value entered for the maximum.

Programmatic Use

To programmatically set the Minimum parameter, use a 1-by-3 vector containing values for the Minimum, Tick Interval, and Maximum parameters, in that order. To use the auto value for the Tick Interval, leave the Tick Interval position in the vector empty or specify -1.

Block Parameter: Limits
Type: 1x3 vector
Default: [0 -1 100]

Maximum — Maximum tick mark value
100 (default) | scalar

Finite, real, double, scalar value specifying the maximum tick mark value for the arc. The maximum must be greater than the value entered for the minimum.
Programmatic Use

To programmatically set the Maximum parameter, use a 1-by-3 vector containing values for the Minimum, Tick Interval, and Maximum parameters, in that order. To use the auto value for the Tick Interval, leave the Tick Interval position in the vector empty or specify -1.

Block Parameter: Limits
Type: 1x3 vector
Default: [0 -1 100]

Tick Interval — Interval between major tick marks
auto (default) | scalar

Finite, real, positive, integer, scalar value specifying the interval of major tick marks on the arc. When set to auto, the block automatically adjusts the tick interval based on the minimum and maximum values.

Programmatic Use

To programmatically set the Tick Interval parameter, use a 1-by-3 vector containing values for the Minimum, Tick Interval, and Maximum parameters, in that order. To use the auto value for the Tick Interval, leave the Tick Interval position in the vector empty or specify -1.

Block Parameter: Limits
Type: 1x3 vector
Default: [0 -1 100]

Scale Colors — Color indications on gauge arc
colors for arc ranges

Color specifications for ranges on the arc. Press the + button to add a color. For each color added, specify the minimum and maximum values of the range where you want to display that color.

Programmatic Use

To programmatically specify the Scale Colors parameter, use an array of structures with fields:

- Min — Minimum value for the color range on the scale.
- Max — Maximum value for the color range on the scale.
- Color — 1-by-3 vector of double values between 0 and 1 to specify the color for the range of the scale in the form [r g b]
Include a structure in the array for each scale range for which you want to specify a color.

```matlab
range1.Min = 0;
range1.Max = 10;
range1.Color = [0 0 1];
range2.Min = 10;
range2.Max = 15;
range2.Color = [0 1 0];
scaleRanges = [range1 range2];
```

**Block Parameter:** ScaleColors  
**Type:** structure array  
**Default:** 0x1 struct array

**Label — Block label position**  
**Top** (default) | **Bottom** | **Hide**

Position of the block label. When the block is connected to an element in the model, the label is the name of the connected element.

**Programmatic Use**  
**Block Parameter:** LabelPosition  
**Type:** character vector  
**Values:** 'Top' | 'Bottom' | 'Hide'  
**Default:** 'Top'

**Block Characteristics**

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>double</th>
<th>enumerated</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.
Dashboard blocks are ignored for code generation.

See Also
Custom Gauge | Gauge | Linear Gauge | Quarter Gauge

Topics
“Tune and Visualize Your Model with Dashboard Blocks”
“Decide How to Visualize Simulation Data”

Introduced in R2015a
Hit Crossing

Detect crossing point

Library:
- Simulink / Discontinuities
- Simulink / Messages & Events
- HDL Coder / Discontinuities
- SimEvents

Description

The Hit Crossing block detects when the input reaches the Hit crossing offset parameter value in the direction specified by the Hit crossing direction property.

You can configure the block to output a 1 or 0 signal or a SimEvents® message. See “Output” on page 1-944 for more information.

Ports

Input

Port_1 — Input signal
scalar | vector

Input signal that the block detects when it reaches the offset in the specified direction.

Data Types: double

Output

Port_1 — Output signal
scalar | vector | SimEvents message

Output indicating if the input signal crossed the hit offset. This port is only visible when you select the Show output port parameter check box.

Signal Output
If you select the **Show output port** check box and set the **Output type** parameter to **Signal**, the block output indicates when the crossing occurs.

- If the input signal is exactly the value of the offset value after the hit crossing is detected in the specified direction, the block continues to output a value of 1.
- If the input signals at two adjacent points brackets the offset value, the block outputs a value of 1 at the second time step.
- If the **Show output port** check box is **not** selected, the block ensures that the simulation finds the crossing point but does not generate output.
- If the initial signal is equal to the offset value, the block outputs 1 only if the **Hit crossing direction** property is set to **either**.
- If Boolean logic signals are enabled, then the output is a **Boolean**.

**SimEvents Message Output**

The Hit Crossing block can also output a SimEvents message when the **Output Type** is set to **Message**.

- If the input signal crosses the offset value in the specified direction, the block outputs a message.
- If the input signal reaches the offset value in the specified direction and remains there, block outputs one message at the hit time and one message when the signal leaves the offset value.
- If the initial input signal is equal to the offset value, the block outputs a message with **Crossing Type value** None only if the **Hit crossing direction** is set to **either**.

The SimEvents message output signal is a **struct** with four fields.

**CrossingType — Direction of zero-crossing**

None | NegativeToPositive | NegativeToZero | ZeroToPositive | PositiveToNegative | PositiveToZero | ZeroToNegative

This field shows the direction in which the signal crosses the **Hit crossing offset** value. Negative, Zero, and Positive are defined relative to the offset value. The data type is **slHitCrossingType** which is an enumerated data type. See “Use Enumerated Data in Simulink Models” for more information. For example, if **HitCrossingOffset** is set to 2, a rising signal crossing this offset value would be recorded as a **NegativeToPositive** hit crossing.
**Note** A hit crossing is recorded based on the **Hit crossing direction** setting. In other words, if you set **Hit crossing direction** to detect a falling hit crossing, a **NegativeToPositive** hit is not recorded.

**Note** In a SimEvents block, if the **Crossing Type** of an entity is a **NegativeToPositive** hitcrossing then `entity.CrossingType == slHitCrossingType.NegativeToPositive` returns logical 1 (true).

If the signal reaches the `HitCrossingOffset` value and holds it, a single **NegativeToZero** or **PositiveToZero**, depending on the direction, hit is registered at the time of the hit crossing.

Data Types: `slHitCrossingType`

**Index — Index of the input signal at which the hit crossing event occurs**

nonnegative integer

For `n` signals being passed to the Hit Crossing block, this field denotes which signal had a hit crossing event. For a matrix input, this field follows MATLAB linear indexing. See “Array Indexing” (MATLAB).

Data Types: `uint32`

**Time — Time of hit crossing event**

real, finite

Time $T$ of the hit crossing event.

Data Types: `double`

**Offset — Hit crossing value for detection**

0 (default) | real values

Hit crossing offset value as specified by the “Hit crossing offset” on page 1-0 parameter.

Data Types: `double`

Data Types: `double` | `Boolean` | `struct`
Note If the SimEvents message output signal crosses model reference boundaries or is used as an input to a Stateflow chart, you need to create a bus object for the message. See “Tips” on page 1-949.

Parameters

Hit crossing offset — Hit crossing value for detection
0 (default) | real values

Specify the value the block detects when the input crosses in the direction specified by Hit crossing direction.

Programmatic Use
Block Parameter: HitCrossingOffset
Type: character vector
Values: real values
Default: '0'

Hit crossing direction — Input signal direction to hit crossing
either (default) | falling | rising

Direction from which the input signal approaches the hit crossing offset for a crossing to be detected.

When set to either, the block serves as an almost equal block, useful in working around limitations in finite mathematics and computer precision. Used for these reasons, this block might be more convenient than adding logic to your model to detect this condition.

When the Hit crossing direction property is set to either and the model uses a fixed-step solver, the block has the following behavior. If the output signal is 1, the block sets the output signal to 0 at the next time step, unless the input signal equals the offset value.

Programmatic Use
Block Parameter: HitCrossingDirection
Type: character vector
Values: 'either' | 'rising' | 'falling'
Default: 'either'

Show output port — Display an output port
off (default) | on
If selected, create an output port on the block icon.

**Programmatic Use**

**Block Parameter:** `ShowOutputPort`  
**Type:** character vector  
**Values:** 'off' | 'on'  
**Default:** 'on'

**Output type — Choose signal or message output**  
signal (default for Simulink) | message (default for SimEvents)

When **Output type** is set to **Signal**, the output signal is set to one whenever the input signal crosses the **Hit crossing offset** value in the **Hit crossing direction** and is zero at other times.

When the **Output type** is set to **Message**, the output signal becomes a SimEvents message.

**Programmatic Use**

**Block Parameter:** `HitCrossingOutputType`  
**Type:** character vector  
**Values:** 'Signal' | 'Message'  
**Default:** 'Signal'

## Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>double</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>yes</td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>no</td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>yes</td>
</tr>
</tbody>
</table>
Tips

If the Hit Crossing block is configured to output a SimEvents message and the output signal:

- Crosses into or out of a referenced model
- Is fed to the input of a Stateflow chart

then you need to create a bus object for the message signal. In the MATLAB Command Window, run Simulink.createHitCrossMessage to check for and, if needed, create a hit crossing message bus object in the base workspace.

Set the data type of the corresponding port to Bus: HitCrossMessage.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

Not recommended for production code.

Does not support non-floating data type for ert targets.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture
This block has a single, default HDL architecture.
HDL Block Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConstrainedOutputPipeline</td>
<td>Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. For more details, see “ConstrainedOutputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td>InputPipeline</td>
<td>Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “InputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td>OutputPipeline</td>
<td>Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “OutputPipeline” (HDL Coder).</td>
</tr>
</tbody>
</table>

Restriction

The Hit crossing direction must be rising or falling.

HDL code generation is not supported when the Output Type is set to Message.

See Also

“Zero-Crossing Detection” | “Implement logic signals as Boolean data (vs. double)”

Introduced before R2006a
IC

Set initial value of signal

Library: Simulink / Signal Attributes

Description

The IC block sets the initial condition of the signal at its input port, for example, the value of the signal at the simulation start time ($t_{\text{start}}$). To do so, the block outputs the specified initial condition when you start the simulation, regardless of the actual value of the input signal. Thereafter, the block outputs the actual value of the input signal.

The IC block is useful for providing an initial guess for the algebraic state variables in a loop. For more information, see “Algebraic Loop Concepts”.

Behavior for Nonzero Sample Time Offset

If an IC block has a nonzero sample time offset ($t_{\text{offset}}$), the IC block outputs its initial value at time $t$,

$$t = n \times t_{\text{period}} + t_{\text{offset}}$$

where $n$ is the smallest integer such that $t \geq t_{\text{start}}$.

That is, the IC block outputs its initial value the first-time blocks with sample time $[t_{\text{period}}, t_{\text{offset}}]$ execute, which can be after $t_{\text{start}}$. 
Ports

Input

**Port_1 — Input signal**

scalar | vector | matrix | N-D array

Input signal, specified as a scalar, vector, matrix, or N-D array. The block sets the initial condition of this signal to the *Initial value* you specify.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus

Output

**Port_1 — Output signal**

scalar | vector | matrix | N-D array

Output signal provided as the *Initial value* you specify, followed by the actual values of the input signal. See “Behavior for Nonzero Sample Time Offset” on page 1-951 for more information.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus

Parameters

**Initial value — Initial value**

1 (default) | real, finite scalar, vector, matrix, or N-D array

Specify the initial value of the input signal as a finite, real-valued scalar, vector, matrix, or N-D array. The value must be a scalar, or have the same dimensions as the input signal.

**Programmatic Use**

**Block Parameter:** Value
**Type:** character vector
**Values:** scalar | vector | matrix | N-D array
**Default:** '1'
Sample time — Specify sample time as a value other than -1
-1 (default) | scalar | vector

Specify the sample time as a value other than -1. For more information, see “Specify Sample Time”.

Dependencies

This parameter is not visible unless it is explicitly set to a value other than -1. To learn more, see “Blocks for Which Sample Time Is Not Recommended”.

Programmatic Use

Block Parameter: SampleTime
Type: character vector
Values: scalar or vector
Default: '-1'

Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>double</th>
<th>enumerated</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.
Not recommended for production code. Relates to resource limits and restrictions on speed and memory often found in embedded systems. Generated code can contain dynamic allocation and freeing of memory, recursion, additional memory overhead, and widely varying execution times. While the code is functionally valid and generally acceptable in resource-rich environments, smaller embedded targets often cannot support such code. Usually, blocks evolve toward being suitable for production code. Thus, blocks suitable for production code remain suitable.

**Fixed-Point Conversion**

Design and simulate fixed-point systems using Fixed-Point Designer™.

**See Also**

**Topics**

“What Is Sample Time?”

“Algebraic Loop Concepts”

**Introduced before R2006a**
If

Select subsystem execution using logic similar to if-else statement

Library: Simulink / Ports & Subsystems

Description

The If block, along with If Action Subsystem blocks containing an Action Port block, implements if-else logic to control subsystem execution. For an example using the If block, see If Action Subsystems.

Limitations

The If block has the following limitations:
• It does not support tunable parameters. Values for an if or elseif expression cannot be tuned during a simulation in normal or accelerator mode, or when running generated code.

To implement tunable if-else expressions, tune the expression outside the If block. For example, use the Relational Operator block to evaluate the expression outside of the If block or add the tunable parameter as an input to the If block.

• It does not support custom storage classes. See “Apply Built-In and Customized Storage Classes to Data Elements” (Embedded Coder).

• The If expression and ElseIf expressions cannot accept certain operators, such as +, -, *, and /

### Ports

**Input**

Logical operands — Values for evaluating logical expressions

- scalar | vector

Inputs $u_1, u_2, \ldots, u_n$ must have the same data type. The inputs cannot be of any user-defined type, such as an enumerated type.

The If block does not directly support fixed-point data types. However, you can use the Compare To Constant block to work around this limitation. See Support for Fixed-Point Data Type in If Action Subsystems.

The If block does not support a boolean data type. If you want to input a boolean signal to an If block, convert the signal to an integer type using the Data Type Conversion block.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32

**Output**

Action — Action signal for an If Action Subsystem block

- scalar

Outputs from the if, else, and elseif ports are action signals to If Action Subsystem blocks.
Support for Fixed-Point Data Type

Parameters

Number of inputs — Specify number of input ports

1 (default) | integer

1

Specify one input port.

integer

Specify the number of input ports. Block ports are labeled with a 'u' character followed by a number, 1, 2, ..., n, where n equals the number of inputs that you specify.

Programmatic Use

Block Parameter: NumInputs

Type: character vector

Values: '1' | '<integer>'

Default: '1'

if expression — Specify logical expression

u1 > 0 (default) | logical expression

The If Action Subsystem attached to the if port executes when its associated expression evaluates to true.

u1 > 0

Specify sending an action signal on the output port when the input u1 is greater than 0.

logical expression

Specify logical expression. This expression appears on the If block adjacent to the if output port.

The expression can include only the operators <, <=, ==, ~=, >, >=, &, |, ~, (), unary-minus. Operators such as +, -, *, /, and ^ are not allowed. The expression must not contain data type expressions, for example, int8(6), and must not reference workspace variables whose data type is other than double or single.
Programmatic Use
Block Parameter: IfExpression
Type: character vector
Values: 'u1 > 0' | '<logical expression>'
Default: 'u1 > 0'

Elseif expressions — Specify logical expression
empty (default) | list of logical expressions

The If Action Subsystem attached to an elseif port executes when its expression evaluates to true and all if and elseif expressions are false.

empty
  Logical expressions not specified.
list of logical expressions
  Specify a list of logical expressions delimited by commas. The expressions appear on the If block below the if port and above the else port when you select the Show else condition check box.

  Expressions can include only the operators <, <=, ==, ~==, >, >=, &, |, ~, (), unary-minus. Operators such as +, -, *, /, and ^ are not allowed. The expressions must not contain data type expressions, for example, int8(6), and must not reference workspace variables whose data type is other than double or single.

Programmatic Use
Block Parameter: ElseIfExpressions
Type: character vector
Values: ' ' | '<list of logical expressions>'
Default: ' '

Show else condition — Control display of else port
on (default) | off

When the if port and all elseif port expressions are false, the else port sends an action signal to execute the attached If Action Subsystem block.

☑ on
  Display else port.
☐ off
  Hide else port.
Programmatic Use
Block Parameter: ShowElse
Type: character vector
Values: 'on' | 'off'
Default: 'on'

Enable zero-crossing detection — Control zero-crossing detection
on (default) | off

Control zero-crossing detection.

☑ on
  Detect zero crossings.

☐ off
  Do not detect zero crossings.

Programmatic Use
Block Parameter: ZeroCross
Type: character vector
Values: 'on' | 'off'
Default: 'on'

Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>double</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also

Blocks
Action Port | If Action Subsystem | Subsystem

Topics
Select Subsystem Execution

Introduced before R2006a
If Action Subsystem

Subsystem whose execution is enabled by an If block
Library: Simulink / Ports & Subsystems

Description

The If Action Subsystem block is a Subsystem block preconfigured as a starting point for creating a subsystem whose execution is controlled by an If block. The If block evaluates a logical expression and then, depending on the result of the evaluation, outputs an action signal.

Simulink ignores a priority set on an If Action Subsystem block. Instead, set the priority on the If block that initiates execution of the subsystem.

Merge signals from If Action Subsystem blocks

This example shows how to merge signals controlled by an If block. The If block selects the execution of an If Action Subsystem block from a set of subsystems. Regardless of which subsystem the If block selects, you can create a single signal with a Merge block. Open model
All blocks in an If Action Subsystem block must execute at the same rate as the driving If block. You can satisfy this requirement by setting the sample time parameter for each block to either inherited (-1) or the same value as the If block sample time.

**Support for Fixed-Point Data Type**

The If block does not directly support fixed-point data types. However, you can use the Compare To Constant block to work around this limitation.

Consider the following floating-point model without fixed-point data types:
In this model, the If Action Subsystem blocks use their default configurations. The simulation parameters are set to their default values except for the parameters listed in the following table.

<table>
<thead>
<tr>
<th>Configuration Parameter Pane</th>
<th>Parameter</th>
<th>Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>Solver</td>
<td>Start time</td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>Stop time</td>
<td>1.0</td>
</tr>
<tr>
<td></td>
<td>Type</td>
<td>Fixed-step</td>
</tr>
<tr>
<td></td>
<td>Solver</td>
<td>discrete (no continuous states)</td>
</tr>
<tr>
<td></td>
<td>Fixed-step size</td>
<td>0.1</td>
</tr>
</tbody>
</table>

The block parameters are set to their default values except for the parameters listed in the following table.

<table>
<thead>
<tr>
<th>Block</th>
<th>Parameter</th>
<th>Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>Repeating Sequence Stair</td>
<td>Vector of output values</td>
<td>[-2 -1 1 2].'</td>
</tr>
</tbody>
</table>
For this model, when input \( u_1 \) is greater than 0 or input \( u_2 \) is greater than 0.5, the output is 4. Otherwise, the output is -4. The Scope block displays the output from the Merge block with inputs \( u_1 \), and \( u_2 \).
You can implement this block diagram as a model with fixed-point data types:
The Repeating Sequence Stair blocks output fixed-point data types.

The Compare To Constant blocks implement two parts of the **If expression** that is used in the If block in the floating-point version of the model, \((u1 > 0)\) and \((u2 > 0.5)\). The OR operation, \((u1|u2)\), can still be implemented inside the If block. For a fixed-point model, the expression must be partially implemented outside of the If block as it is in this model.

The block and simulation parameters for the fixed-point model are the same as for the floating-point model with the following exceptions and additions:

<table>
<thead>
<tr>
<th>Block</th>
<th>Parameter</th>
<th>Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compare To Constant</td>
<td><strong>Operator</strong></td>
<td>&gt;</td>
</tr>
<tr>
<td></td>
<td><strong>Constant value</strong></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td><strong>Output data type mode</strong></td>
<td>Boolean</td>
</tr>
<tr>
<td></td>
<td><strong>Enable zero-crossing</strong></td>
<td>off</td>
</tr>
<tr>
<td></td>
<td>detection</td>
<td></td>
</tr>
<tr>
<td>Compare To Constant1</td>
<td><strong>Operator</strong></td>
<td>&gt;</td>
</tr>
<tr>
<td></td>
<td><strong>Constant value</strong></td>
<td>0.5</td>
</tr>
<tr>
<td></td>
<td><strong>Output data type mode</strong></td>
<td>Boolean</td>
</tr>
<tr>
<td>Block</td>
<td>Parameter</td>
<td>Setting</td>
</tr>
<tr>
<td>-------</td>
<td>---------------------------------</td>
<td>-----------</td>
</tr>
<tr>
<td></td>
<td>Enable zero-crossing detection</td>
<td>off</td>
</tr>
<tr>
<td>If</td>
<td>Number of inputs</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>If expression</td>
<td>u1</td>
</tr>
</tbody>
</table>

## Ports

### Input

**In — Signal input to a subsystem block**

scalar | vector | matrix

Placing an Inport block in a subsystem block adds an external input port to the block. The port label matches the name of the Inport block.

Use Inport blocks to get signals from the local environment.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus

**Action — Control signal input to a subsystem block**

scalar | vector | matrix

Placing an Action Port block in a subsystem block adds an external input port to the block and changes the block to an If Action Subsystem.

Dot-dash lines from a Switch Case block to an Switch Case Action Subsystem block represent action signals. An action signal is a control signal connected to the action port of a Switch Case Action Subsystem block. A message on the action signal initiates execution of the subsystem.

Data Types: action

### Output

**Out — Signal output from a subsystem**

scalar | vector | matrix
Placing an Outport block in a subsystem block adds an output port from the block. The port label on the subsystem block is the name of the Outport block.

Use Outport blocks to send signals to the local environment.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus

### Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>bus</th>
<th>double</th>
<th>enumerated</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
<th>string</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Direct Feedthrough</th>
<th>no</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multidimensional Signals</td>
<td>limited</td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>limited</td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
</tr>
</tbody>
</table>

a. Actual data type or capability support depends on block implementation.

### Extended Capabilities

**C/C++ Code Generation**
Generate C and C++ code using Simulink® Coder™.

Actual data type or capability support depends on block implementation.

**Fixed-Point Conversion**
Design and simulate fixed-point systems using Fixed-Point Designer™.
See Also

Blocks
Action Port | If | Subsystem

Topics
Select Subsystem Execution

Introduced before R2006a
In Bus Element, Bus Element In

Select signal that connects to input port

Library:  
- Simulink / Ports & Subsystems
- Simulink / Sources
- HDL Coder / Ports & Subsystems
- HDL Coder / Sources

Description

The In Bus Element block, also known as the Bus Element In block, combines the functionality of an Inport block and a Bus Selector block. This block is of the Inport block type and all In Bus Element blocks that use the same port share a Block Parameters dialog box.

You can use an In Bus Element block to select these signals associated with the port:

- For a port that receives a bus, you can select a bus element by specifying a signal name or signal hierarchy in the block label.
- For a port that receives a bus, you can select the entire bus by deleting the signal name in the block label. The block label displays only the port name.
- For a port that receives a nonbus signal, you can select the nonbus signal by deleting the signal name in the block label. The block label displays only the port name.

To select multiple signals from a bus, create multiple In Bus Element blocks. You can create In Bus Element blocks for each selected signal in the Block Parameters dialog box by clicking the Add blocks for selected signals button.

You can load simulation input data for a group of root-level In Bus Element blocks using a structure of timeseries, timetable, or SimulationDatastore objects. In Bus Element blocks interpolate loaded data for all data types except string, enumerated, and fixed point.

Specify a Signal

To change the signal associated with a bus element port block, you must edit the label next to the block by double-clicking the text. Dot notation indicates signal hierarchy
within the bus. To reduce the size of the label, you can display only the signal name by selecting **Compact Notation** from the ellipsis menu that appears when you select the block. To show the signal hierarchy again, you can select **Expanded Notation** from the menu.

### Ports

### Output

**Port_1 — Selected input signal**

Selected input bus or bus element that is associated with an external subsystem or model port.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `Boolean` | `fixed point` | `enumerated` | `bus`

### Parameters

**Port name — Name of input port**

Specify a port name that is not already in use by another block or port. The name appears on the parent Subsystem or Model block. The name also appears next to the block. Multiple blocks can access the same port.

Programmatic Use

**Block Parameter:** PortName  
**Type:** character vector  
**Values:** `'InBus'` | `'<port name>'`  
**Default:** `'InBus'`

**Port number — Position of port on parent block**

Specify the order in which the port that corresponds to the block appears on the parent Subsystem or Model block.
• If you add a block that creates another port, the port number is the next available number.
• Deleting all blocks associated with a port deletes the port. Other ports are renumbered so that they are sequential and do not skip any numbers.
• Specifying a port number that exceeds the number of ports creates a port for that number and for any skipped sequential numbers.

**Programmatic Use**

**Block Parameter:** Port
**Type:** character vector
**Values:** real integer
**Default:** '1'

**Set color — Specify block background color**

Specify the background color for bus element port blocks. This specification sets the color of blocks associated with selected bus elements, or of all blocks associated with the port if you do not select any bus elements.

You can specify Black, White, Red, Green, Blue, Cyan, Magenta, Yellow, Gray, Light Blue, Orange, Dark Green, and More Colors.

**Programmatic Use**

**Block Parameter:** BackgroundColor
**Type:** character vector
**Values:** 'black' | 'white' | 'red' | 'green' | 'blue' | 'cyan' | 'magenta' | 'yellow' | 'gray' | 'lightBlue' | 'orange' | 'darkGreen' | '[r,g,b]' where r, g, and b, are the red, green, blue values of the color in the range 0.0 to 1.0
**Default:** 'black'

**Filter — Filter set of displayed signals**

Specify a search term to use for filtering a long list of bus elements. Do not enclose the search term in quotation marks. The filter does a partial string search and supports regular expressions. To use a regular expression character as a literal, include an escape character (\). For example, to use a question mark, type sig\?1.
Signal Attributes

To specify signal attributes, click the pencil button or the specified attributes next to a signal name.

**Data Type — Data type**

Inherit: auto (default) | double | single | int8 | uint8 | int16 | uint16 | int32 | uint32 | int64 | uint64 | fixdt(1,16) | fixdt(1,16,0) | fixdt(1,16,2^0,0) | Enum: <class name> | Bus: <object name> | <data type expression>

Specify the data type. The type can be inherited, specified directly, or expressed as a data type object such as Simulink.NumericType. Do not specify a bus object as the expression.

**Dimensions — Signal dimensions**

-1 (default) | integer | [integer, integer]

Specify the dimensions of a signal.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>The signal can have any dimensions.</td>
</tr>
<tr>
<td>N</td>
<td>The signal must be a vector of size N.</td>
</tr>
<tr>
<td>[R C]</td>
<td>The signal must be a matrix having R rows and C columns.</td>
</tr>
</tbody>
</table>

**Programmatic Use**

**Block Parameter:** PortDimensions  
**Type:** character vector  
**Values:** '-' | integer | [integer, integer]  
**Default:** '-'

**Dims Mode — Allow variable-size signals**

Inherit (default) | Fixed | Variable

Specify the type of signals allowed. To allow variable-size and fixed-size signals, select Inherit. To allow only variable-size signals, select Variable. To not allow variable-size signals, select Fixed.

**Dependencies**

When the signal is variable-sized, the **Dimensions** parameter specifies the maximum dimensions of the signal.
**Command-Line Information**
**Parameter:** VarSizeSig  
**Type:** character vector  
**Value:** 'Inherit' | 'No' | 'Yes'  
**Default:** 'Inherit'

**Unit — Physical unit of the signal**

*inherit* (default) | <Enter unit>*

Specify the physical unit of the signal. For a list of supported units, see Allowed Unit Systems.

**Programmatic Use**
**Block Parameter:** Unit  
**Type:** character vector  
**Values:** 'inherit' | '<Enter unit>'  
**Default:** 'inherit'

**Sample time — Specify sample time**

-1 (default) | scalar

Specify the discrete interval between sample time hits or specify another appropriate sample time such as continuous or inherited.

By default, the signal inherits its sample time. To set a different sample time, enter a valid sample time based on the table in “Types of Sample Time”.

**Programmatic Use**
**Block Parameter:** SampleTime  
**Type:** character vector  
**Values:** scalar  
**Default:** '-1'

**Complexity — Signal type**

*auto* (default) | *real* | *complex*

Specify the numeric type of the signal. To choose the numeric type of the signal, select *auto*. Otherwise, choose a real or complex signal type.

**Programmatic Use**
**Block Parameter:** SignalType  
**Type:** character vector
**Values:** 'auto' | 'real' | 'complex'
**Default:** 'auto'

**Minimum — Minimum value**

[] (default) | scalar

Lower value of the range that Simulink checks.

This number must be a finite real double scalar value.

Simulink software uses this value to perform:

- Simulation range checking (see “Signal Ranges”).
- Automatic scaling of fixed-point data types.
- Optimization of the code that you generate from the model. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. For more information, see “Optimize using the specified minimum and maximum values” (Simulink Coder).

**Programmatic Use**

**Block Parameter:** OutMin

**Type:** character vector

**Values:** ' [ ] ' | scalar

**Default:** ' [ ] '

**Maximum — Maximum value**

[] (default) | scalar

Upper value of the range that Simulink checks.

This number must be a finite real double scalar value.

Simulink software uses this value to perform:

- Simulation range checking (see “Signal Ranges”).
- Automatic scaling of fixed-point data types.
- Optimization of the code that you generate from the model. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. For more information, see “Optimize using the specified minimum and maximum values” (Simulink Coder).
**Programmatic Use**

*Block Parameter:* OutMax

*Type:* character vector

*Values:* '[]' | scalar

*Default:* '[]'

**Block Characteristics**

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>bus</th>
<th>double</th>
<th>enumerated</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
<th>string</th>
</tr>
</thead>
</table>

*Direct Feedthrough*  

no

*Multidimensional Signals*  

yes

*Variable-Size Signals*  

yes

*Zero-Crossing Detection*  

no

**Extended Capabilities**

**C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

**HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

This block can be used to simplify the Subsystem bus interfaces when you use the block in subsystems that generate HDL code, but is not included in the hardware implementation.

To learn more about using buses for HDL code generation, see “Buses” (HDL Coder) and “Use Bus Signals to Improve Readability of Model and Generate HDL Code” (HDL Coder).
Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

See Also

Blocks
Bus Selector | Inport | Out Bus Element

Topics
“Simplify Bus Interfaces”
“Types of Composite Signals”
“Getting Started with Buses”

Introduced in R2017a
Increment Real World

Increase real-world value of signal by one

**Library:**
- Simulink / Additional Math & Discrete / Additional Math: Increment - Decrement
- HDL Coder / Math Operations

**Description**

The Increment Real World block increases the real-world value of the signal by one.

Overflows always wrap.

**Ports**

**Input**

**Port_1 — Input signal**
scalar | vector | matrix

Input signal, specified as a scalar, vector, or matrix.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

**Output**

**Port_1 — Output signal**
scalar | vector | matrix

Output is the real-world value of the input signal increased by one. Overflows always wrap. The output has the same data type and dimensions as the input.
Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

### Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>double</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Extended Capabilities

#### C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

The code generator does not explicitly group primitive blocks that constitute a nonatomic masked subsystem block in the generated code. This flexibility allows for more efficient code generation. In certain cases, you can achieve grouping by configuring the masked subsystem block to execute as an atomic unit by selecting the Treat as atomic unit option.

#### HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.
**HDL Architecture**

This block has a single, default HDL architecture.

**HDL Block Properties**

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ConstrainedOutput Pipeline</strong></td>
<td>Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. For more details, see “ConstrainedOutputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td><strong>InputPipeline</strong></td>
<td>Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “InputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td><strong>OutputPipeline</strong></td>
<td>Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “OutputPipeline” (HDL Coder).</td>
</tr>
</tbody>
</table>

**Fixed-Point Conversion**

Design and simulate fixed-point systems using Fixed-Point Designer™.

**See Also**

Decrement Stored Integer | Decrement Time To Zero | Decrement To Zero | Increment Real World

**Topics**

“Fixed-Point Numbers”

**Introduced before R2006a**
Increment Stored Integer

Increase stored integer value of signal by one

**Library:**
- Simulink / Additional Math & Discrete / Additional Math: Increment - Decrement
- HDL Coder / Math Operations

**Description**

The Increment Stored Integer block increases the stored integer value of a signal by one. Floating-point signals also increase by one, and overflows always wrap.

**Ports**

**Input**

**Port_1 — Input signal**
- scalar | vector | matrix

Input signal, specified as a scalar, vector, or matrix.

**Data Types:** single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

**Output**

**Port_1 — Output signal**
- scalar | vector | matrix

Output is the stored integer value of the input signal increased by one. Floating-point signals also increase by one, and overflows always wrap. The output has the same data type and dimensions as the input.
Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>double</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

The code generator does not explicitly group primitive blocks that constitute a nonatomic masked subsystem block in the generated code. This flexibility allows for more efficient code generation. In certain cases, you can achieve grouping by configuring the masked subsystem block to execute as an atomic unit by selecting the Treat as atomic unit option.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.
HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConstrainedOutput Pipeline</td>
<td>Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. For more details, see “ConstrainedOutputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td>InputPipeline</td>
<td>Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “InputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td>OutputPipeline</td>
<td>Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “OutputPipeline” (HDL Coder).</td>
</tr>
</tbody>
</table>

Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

See Also
Decrement Stored Integer | Increment Real World

Topics
“Fixed-Point Numbers”

Introduced before R2006a
Index Vector

Switch output between different inputs based on value of first input

**Library:**
- Simulink / Signal Routing
- HDL Coder / Signal Routing

Description

The Index Vector block is a special configuration of the Multiport Switch block in which you specify one data input and the control input is zero-based. The block output is the element of the input vector whose index matches the control input. For example, if the input vector is \([18 \ 15 \ 17 \ 10]\) and the control input is 3, the element that matches the index of 3 (zero-based) is 10, and that becomes the output value.

To configure a Multiport Switch block to work as an Index Vector block set **Number of data ports** to 1 and **Data port order** to **Zero-based contiguous**.

For more information about the Multiport Switch block, see the Multiport Switch block reference page.

Ports

Input

**Port_1 — Control signal**

*scalar*

Control signal, specified as a scalar. When the control input is not an integer value, the block truncates the value to an integer by rounding to zero.

For information on control signals of enumerated type, see “Guidelines on Setting Parameters for Enumerated Control Port” on page 1-1344 on the Multiport Switch block ref page.
Limitations

- If the control signal is numeric, the control signal cannot be complex.
- If the control signal is an enumerated signal, the block uses the value of the underlying integer to select a data port.
- If the underlying integer does not correspond to a data input, an error occurs.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point

0 or 1 — First data input
scalar | vector

First data input, specified as a scalar or vector. The port is labeled 0 when you set Data port order to Zero-based contiguous, and labeled 1 when you set Data port order to One-based contiguous.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | bus | enumerated | string

Output

Port_1 — Selected data input, based on control signal value
scalar

The block outputs the selected value from the input data vector, according to the control signal value. The output is a scalar.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus

Parameters

Main

Data port order — Type of ordering for data input ports
Zero-based contiguous (default) | One-based contiguous | Specify indices

Specify the type of ordering for your data input ports.
• **Zero-based contiguous** — Block uses zero-based indexing for ordering contiguous data ports. This is the default value of the Index Vector block.

• **One-based contiguous** — Block uses one-based indexing for ordering contiguous data ports. This is the default value of the Multiport Switch block.

• **Specify indices** — Block uses noncontiguous indexing for ordering data ports. This value is supported only for configurations with two or more input data ports.

**Tips**

• When the control port is of enumerated type, select **Specify indices**.

• If you select **Zero-based contiguous** or **One-based contiguous**, verify that the control port is not of enumerated type. This configuration is deprecated and produces an error. You can run the Upgrade Advisor on your model to replace each Multiport Switch block of this configuration with a block that explicitly specifies data port indices. See “Model Upgrades”.

• Avoid situations where the block contains unused data ports for simulation or code generation. When the control port is of fixed-point or built-in data type, verify that all data port indices are representable with that type. Otherwise, the following block behavior occurs:

<table>
<thead>
<tr>
<th>If the block has unused data ports and data port order is:</th>
<th>The block produces:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zero-based contiguous or One-based contiguous</td>
<td>A warning</td>
</tr>
<tr>
<td>Specify indices</td>
<td>An error</td>
</tr>
</tbody>
</table>

**Dependencies**

Selecting **Zero-based contiguous** or **One-based contiguous** enables the **Number of data ports** parameter.

Selecting **Specify indices** enables the **Data port indices** parameter.

**Programmatic Use**

**Block Parameter:** DataPortOrder  
**Type:** character vector  
**Values:** 'Zero-based contiguous' | 'One-based contiguous' | 'Specify indices'  
**Default:** 'Zero-based contiguous'
Number of data ports — Number of data input ports
1 (default) | integer between 1 and 65536

Specify the number of data input ports to the block.

Dependencies

To enable this parameter, set Data port order to Zero-based contiguous or One-based contiguous.

Programmatic Use

Block Parameter: Inputs
Type: character vector
Values: integer between 1 and 65536
Default: '1'

Signal Attributes

Require all data port inputs to have the same data type — Require all inputs to have the same data type
off (default) | on

Select this check box to require that all data input ports have the same data type. When you clear this check box, the block allows data port inputs to have different data types.

Programmatic Use

Block Parameter: InputSameDT
Type: character vector
Values: 'off' | 'on'
Default: 'off'

Output minimum — Minimum output value for range checking
[ ] (default) | scalar

Lower value of the output range that Simulink checks.

Simulink uses the minimum to perform:

- Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters”) for some blocks.
- Simulation range checking (see “Signal Ranges” and “Enable Simulation Range Checking”).
• Automatic scaling of fixed-point data types.
• Optimization of the code that you generate from the model. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. For more information, see “Optimize using the specified minimum and maximum values” (Simulink Coder).

**Note** Output minimum does not saturate or clip the actual output signal. Use the Saturation block instead.

**Programmatic Use**

**Block Parameter:** OutMin

**Type:** character vector

**Values:** ' [ ] ' | scalar

**Default:** ' [ ] '

**Output maximum — Maximum output value for range checking**

[ ] (default) | scalar

Upper value of the output range that Simulink checks.

Simulink uses the maximum value to perform:

• Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters”) for some blocks.

• Simulation range checking (see “Signal Ranges” and “Enable Simulation Range Checking”).

• Automatic scaling of fixed-point data types.

• Optimization of the code that you generate from the model. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. For more information, see “Optimize using the specified minimum and maximum values” (Simulink Coder).

**Note** Output maximum does not saturate or clip the actual output signal. Use the Saturation block instead.
Programmatic Use
Block Parameter: OutMax
Type: character vector
Values: '[ ]' | scalar
Default: '[ ]'

Output data type — Specify the output data type
Inherit: Inherit via internal rule (default) | Inherit: Inherit via back propagation | double | single | int8 | uint8 | int16 | uint16 | int32 | uint32 | int64 | uint64 | fixdt(1,16) | fixdt(1,16,0) | fixdt(1,16,2^0,0) | <data type expression>

Choose the data type for the output. The type can be inherited, specified directly, or expressed as a data type object such as Simulink.NumericType.

When you select an inherited option, the block behaves as follows:

- Inherit: Inherit via internal rule—Simulink chooses a data type to balance numerical accuracy, performance, and generated code size, while taking into account the properties of the embedded target hardware. If you change the embedded target settings, the data type selected by the internal rule might change. It is not always possible for the software to optimize code efficiency and numerical accuracy at the same time. If the internal rule doesn’t meet your specific needs for numerical accuracy or performance, use one of the following options:
  - Specify the output data type explicitly.
  - Explicitly specify a default data type such as fixdt(1,32,16) and then use the Fixed-Point Tool to propose data types for your model. For more information, see fxptdlg.
  - To specify your own inheritance rule, use Inherit: Inherit via back propagation and then use a Data Type Propagation block. Examples of how to use this block are available in the Signal Attributes library Data Type Propagation Examples block.
- Inherit: Inherit via back propagation — Uses the data type of the driving block.

Programmatic Use
Block Parameter: OutDataTypeStr
Type: character vector
**Values:** 'Inherit: Inherit via internal rule' | 'Inherit: Inherit via back propagation' | 'double' | 'single' | 'int8' | 'uint8' | 'int16' | 'uint16' | 'int32' | 'uint32' | 'int64' | 'uint64' | 'fixdt(1,16)' | 'fixdt(1,16,0)' | 'fixdt(1,16,2^0,0)' | '<data type expression>'

**Default:** 'Inherit: Inherit via internal rule'

**Lock output data type setting against changes by the fixed-point tools — Prevent fixed-point tools from overriding data types**

*off* (default) | *on*

Select to lock the output data type setting of this block against changes by the Fixed-Point Tool and the Fixed-Point Advisor. For more information, see “Use Lock Output Data Type Setting” (Fixed-Point Designer).

**Programmatic Use**

**Block Parameter:** `LockScale`

**Type:** character vector

**Values:** 'off' | 'on'

**Default:** 'off'

**Integer rounding mode — Specify the rounding mode for fixed-point operations**

*Floor* (default) | *Ceiling* | *Convergent* | *Nearest* | *Round* | *Simplest* | *Zero*

Choose one of these rounding modes.

**Ceiling**

Rounds both positive and negative numbers toward positive infinity. Equivalent to the MATLAB `ceil` function.

**Convergent**

Rounds number to the nearest representable value. If a tie occurs, rounds to the nearest even integer. Equivalent to the Fixed-Point Designer `convergent` function.

**Floor**

Rounds both positive and negative numbers toward negative infinity. Equivalent to the MATLAB `floor` function.

**Nearest**

Rounds number to the nearest representable value. If a tie occurs, rounds toward positive infinity. Equivalent to the Fixed-Point Designer `nearest` function.
Round

Rounds number to the nearest representable value. If a tie occurs, rounds positive numbers toward positive infinity and rounds negative numbers toward negative infinity. Equivalent to the Fixed-Point Designer round function.

Simplest

Automatically chooses between round toward floor and round toward zero to generate rounding code that is as efficient as possible.

Zero

Rounds number toward zero. Equivalent to the MATLAB fix function.

Programmatic Use

Block Parameter: RndMeth
Type: character vector
Values: 'Ceiling' | 'Convergent' | 'Floor' | 'Nearest' | 'Round' | 'Simplest' | 'Zero'
Default: 'Floor'

See Also

For more information, see “Rounding” (Fixed-Point Designer).

Saturate on integer overflow — Method of overflow action

off (default) | on

Specify whether overflows saturate or wrap.

- off — Overflows wrap to the appropriate value that the data type can represent.

  For example, the number 130 does not fit in a signed 8-bit integer and wraps to -126.

- on — Overflows saturate to either the minimum or maximum value that the data type can represent.

  For example, an overflow associated with a signed 8-bit integer can saturate to -128 or 127.

Tip

- Consider selecting this check box when your model has a possible overflow and you want explicit saturation protection in the generated code.
• Consider clearing this check box when you want to optimize efficiency of your generated code.

Clearing this check box also helps you to avoid overspecifying how a block handles out-of-range signals. For more information, see “Check for Signal Range Errors”.

• When you select this check box, saturation applies to every internal operation on the block, not just the output or result.

• In general, the code generation process can detect when overflow is not possible. In this case, the code generator does not produce saturation code.

**Programmatic Use**

**Block Parameter:** SaturateOnIntegerOverflow  
Type: character vector  
Values: 'off' | 'on'  
Default: 'off'

Allow different data input sizes (Results in variable-size output signal) — Allow input signals with different sizes

off (default) | on

Select this check box to allow input signals with different sizes.

• On — Allows input signals with different sizes, and propagate the input signal size to the output signal. In this mode, the block produces a variable-size output signal.

• Off — Requires that all nonscalar data input signals be the same size.

**Programmatic Use**

**Parameter:** AllowDiffInputSizes  
Type: character vector  
Value: 'on' | 'off'  
Default: 'off'

**Block Characteristics**

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>bus</th>
<th>double</th>
<th>enumerated</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>--------------------------------</td>
<td>-----</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Direct Feedthrough</strong></td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Multidimensional Signals</strong></td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Variable-Size Signals</strong></td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Zero-Crossing Detection</strong></td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Extended Capabilities

#### C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

#### HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

The Index Vector block is a Multiport Switch block with **Number of data ports** set to 1. For HDL code generation information, see Multiport Switch.

#### PLC Code Generation
Generate Structured Text code using Simulink® PLC Coder™.

#### Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.
See Also
Multiport Switch | Switch

Topics
“Variable-Size Signal Basics”

Introduced before R2006a
Initialize Function

Executes contents on a model initialize event

**Library:** Simulink / User-Defined Functions

---

**Description**

The Initialize Function block is a pre-configured subsystem block that executes on a model initialize event. By default, the Initialize Function block includes an Event Listener block with **Event** set to Initialize, a Constant block with **Constant value** set to 0, and a State Writer block.

Replace the Constant block with blocks that generate the state value for the State Writer block.

Conditionally executed subsystem blocks with output ports fully supported within Terminate Function blocks

For a list of unsupported blocks and features, see “Initialize, Reset, and Terminate Function Limitations”.

The input and output ports of a component containing Initialize Function and Terminate Function blocks must connect to input and output port blocks.
The code generated from this block is part of the `model_initialize` function that is called once at the beginning of model execution.

**Block Characteristics**

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>bus</th>
<th>double</th>
<th>enumerated</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
<th>string</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>limited</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

a. Actual data type or capability support depends on block implementation.

**Extended Capabilities**

**C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

Actual code generation support depends on block implementation.

**Fixed-Point Conversion**

Design and simulate fixed-point systems using Fixed-Point Designer™.

Actual data type support depends on block implementation.

**See Also**

Event Listener | Reset Function | State Reader | State Writer | Terminate Function
Topics

“Using Initialize, Reset, and Terminate Functions”
“Create Test Harness to Generate Function Calls”
“Generate Code That Responds to Initialize, Reset, and Terminate Events” (Simulink Coder)
Inport

Create input port for subsystem or external input

Library:
- Simulink / Commonly Used Blocks
- Simulink / Ports & Subsystems
- Simulink / Sources
- HDL Coder / Commonly Used Blocks
- HDL Coder / Ports & Subsystems
- HDL Coder / Sources

Description

Inport blocks link signals from outside a system into the system.

Simulink software assigns Inport block port numbers according to these rules:

- It automatically numbers the Inport blocks within a top-level system or subsystem sequentially, starting with 1.
- If you add an Inport block, the label is the next available number.
- If you delete an Inport block, other port numbers are automatically renumbered to ensure that the Inport blocks are in sequence and that no numbers are omitted.
- If you copy an Inport block into a system, its port number is not renumbered unless its current number conflicts with an inport already in the system. If the copied Inport block port number is not in sequence, renumber the block. Otherwise, you get an error message when you run the simulation or update the block diagram.

Inport Blocks in a Top-Level System

You can use an Inport block in a top-level system to:

- Supply external inputs from the workspace using one of these approaches. If no external outputs are supplied, then the default output is the ground value.
  - Use the Configuration Parameters > Data Import/Export > Input parameter. See “Load Data to Root-Level Input Ports”.

1-998
Tip To import many signals to root-level input ports, consider using the Root Inport Mapper tool. For more information, see “Map Data Using Root Inport Mapper Tool”.

- Use the `ut` argument of the `sim` command to specify the inputs.
- Provide a means for perturbation of the model by the `linmod` and `trim` analysis functions.
- Use Inport blocks to inject inputs into the system. See “Linearizing Models”.
- To load logged signal data using root Inport blocks, use the `createInputDataset` function to create a `Dataset` object that contains elements corresponding to root-level Inport blocks.

**Inport Blocks in a Subsystem**

Inport blocks in a subsystem represent inputs to the subsystem. A signal arriving at an input port on a Subsystem block flows out of the associated Inport block in that subsystem. The Inport block associated with an input port on a Subsystem block is the block whose `Port number` parameter matches the relative position of the input port on the Subsystem block. For example, the Inport block whose `Port number` parameter is 1 gets its signal from the block connected to the topmost port on the Subsystem block.

If you renumber the `Port number` of an Inport block, the block becomes connected to a different input port. The block continues to receive its signal from the same block outside the subsystem.

Inport blocks inside a subsystem support signal label propagation, but root-level Inport blocks do not.

**Tip** For models that include bus signals composed of many bus elements, consider using In Bus Element and Out Bus Element blocks. These blocks:

- Reduce signal line complexity and clutter in a block diagram.
- Make it easier to change the interface incrementally.
- Allow access to a bus element closer to the point of usage, avoiding the use of a Bus Selector and Goto block configuration.

The In Bus Element block is of block type Inport and the Out Bus Element block is of block type Outport.
Creating Duplicate Inports

You can create any number of duplicates of an Inport block. The duplicates are graphical representations of the original intended to simplify block diagrams by eliminating unnecessary lines. The duplicate has the same port number, properties, and output as the original.

To create a duplicate of an Inport block:

1. In the block diagram, select the unconnected Inport block that you want to duplicate.
2. Press and hold the Ctrl key and drag the block.
3. Release the mouse and then select Duplicate from the context menu.

Connecting Buses to Root-Level Inports

If you want a root-level Inport of a model to produce a bus signal, set the Data type parameter to the name of a bus object that defines the bus that the Inport produces. For more information, see “Specify Bus Properties with Bus Objects”.

Ports

Output

Port_1 — Inport signal

scalar | vector

Input signal that flows through the inport into the system.

You can use a subsystem inport to supply fixed-point data in a structure or any other format.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus
Parameters

Main

Port number — Position of port on parent block
1 (default) | real integer

Specify the order in which the port that corresponds to the block appears on the parent Subsystem or Model block.

- If you add a block that creates another port, the port number is the next available number.
- Deleting all blocks associated with a port deletes the port. Other ports are renumbered so that they are sequential and do not skip any numbers.
- Specifying a port number that exceeds the number of ports creates a port for that number and for any skipped sequential numbers.

Programmatic Use
Block Parameter: Port
Type: character vector
Values: real integer
Default: '1'

Icon display — Icon display
Port number (default) | Signal name | Port number and signal name

Specify the information displayed on the block icon.

Programmatic Use
Block Parameter: IconDisplay
Type: character vector
Values: 'Signal name' | 'Port number' | 'Port number and signal name'
Default: 'Port number'

Latch input by delaying outside signal — Latch signal by delay
off (default) | on

Select to specify the block outputs the value of the input signal at the previous time step.

Selecting this check box enables Simulink to resolve data dependencies among triggered subsystems that are part of a loop.
The Inport block indicates that this option is selected by displaying <Lo>.

![Diagram](image)

**Dependency**

Enabled in a triggered subsystem.

**Programmatic Use**

**Block Parameter:** LatchByDelaying OutsideSignal

**Type:** character vector

**Values:** 'on' | 'off'

**Default:** 'off'

**Latch input for feedback signals of function-call subsystem outputs**

— **Latch signal from changing**

off (default) | on

Select to specify the block latches the value of the input to this subsystem and prevents this value from changing during the execution of the subsystem. For a single function call that is branched to invoke multiple function-call subsystems, this option breaks a loop formed by a signal fed back from one of these function-call subsystems into the other. This option prevents any change to the values of a feedback signal from a function-call subsystem that is invoked during the execution of this subsystem.

The Inport block indicates that this option is selected by displaying <Li>.
**Dependency**

Enabled when Inport block is in a function-call subsystem.

**Programmatic Use**

**Block Parameter:** LatchInputFor FeedbackSignals  
**Type:** character vector  
**Values:** 'on' | 'off'  
**Default:** 'off'

**Interpolate data — Interpolate output data**

When loading data from the workspace to a root-level Inport block, specify whether the block linearly interpolates and extrapolates output at time steps for which no corresponding data exists.

To load discrete signal data from the workspace, in the Inport block dialog box:

1. Set the **Sample time** parameter to a discrete value, such as 2.
2. Clear the **Interpolate data** parameter.

Specifying the discrete sample time causes the simulation to have hit times exactly at those instances when the discrete data is sampled. You specify the data values, not time values.

Turning interpolation off avoids unexpected data values at other simulation time points as a result of double precision arithmetic processing. For more information, see “Load Data to Test a Discrete Algorithm”.

Simulink uses the following interpolation and extrapolation:
- For time steps between the first specified data point and the last specified data point — zero-order hold
- For time steps before the first specified data point and after the last specified data point — ground value
- For variable-size signals for time steps before the first specified data point — a NaN is logged for single or double data types and ground for other data types. For time steps after the last specified data point, uses ground values.

**Programmatic Use**

**Block Parameter:** Interpolate  
**Type:** character vector  
**Values:** 'on' | 'off'  
**Default:** 'on'

**Connect Input — Tool to help map signals to inports**

To import, visualize, and map signal and bus data to root-level inports, click this button. The Root Inport Mapper tool displays.

**Dependency**

This button appears only if this block is a root inport block.

**Signal Attributes**

Click the **Show data type assistant** button to display the **Data Type Assistant**, which helps you set the data type attributes. For more information, see “Specify Data Types Using Data Type Assistant”.

**Output function call — Output function-call trigger signal**

off (default) | on

Specify that the input signal outputs a function-call trigger signal.

Select this option if it is necessary for a current model to accept a function-call trigger signal when referenced in the top model.

**Dependency**

Enabled in an asynchronous function call.
**Minimum — Minimum output value**

[] (default) | scalar

Lower value of the output range that Simulink checks.

This number must be a finite real double scalar value.

**Note** If you specify a bus object as the data type for this block, do not set the minimum value for bus data on the block. Simulink ignores this setting. Instead, set the minimum values for bus elements of the bus object specified as the data type. For information on the Minimum property of a bus element, see `Simulink.BusElement`.

Simulink software uses this value to perform:

- Simulation range checking (see “Signal Ranges”).
- Automatic scaling of fixed-point data types.
- Optimization of the code that you generate from the model. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. For more information, see “Optimize using the specified minimum and maximum values” (Simulink Coder).

**Programmatic Use**

**Block Parameter:** `OutMin`  
**Type:** character vector  
**Values:** '[]' | scalar  
**Default:** '[]'

**Maximum — Maximum output value**

[] (default) | scalar

Upper value of the output range that Simulink checks.

This number must be a finite real double scalar value.

**Note** If you specify a bus object as the data type for this block, do not set the maximum value for bus data on the block. Simulink ignores this setting. Instead, set the maximum values for bus elements of the bus object specified as the data type. For information on the Maximum property of a bus element, see `Simulink.BusElement`.
Simulink software uses this value to perform:

- Simulation range checking (see “Signal Ranges”).
- Automatic scaling of fixed-point data types.
- Optimization of the code that you generate from the model. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. For more information, see “Optimize using the specified minimum and maximum values” (Simulink Coder).

**Programmatic Use**

**Block Parameter:** OutMax  
**Type:** character vector  
**Values:** '[ ]' | scalar  
**Default:** '[ ]'

**Data type — Output data type**

Inherit: auto (default) | double | single | int8 | uint8 | int16 | uint16 | int32 | uint32 | int64 | uint64 | fixdt(1,16) | fixdt(1,16,0) | fixdt(1,16,2^0,0) | Enum: <class name> | Bus: <object name> | <data type expression>

Specify the output data type of the external input. The type can be inherited, specified directly, or expressed as a data type object such as Simulink.NumericType. Do not specify a bus object as the expression.

**Lock output data type setting against changes by the fixed-point tools — Prevent fixed-point tools from overriding data types**

off (default) | on

Select to lock the output data type setting of this block against changes by the Fixed-Point Tool and the Fixed-Point Advisor. For more information, see “Use Lock Output Data Type Setting” (Fixed-Point Designer).

**Programmatic Use**

**Block Parameter:** LockScale  
**Type:** character vector  
**Values:** 'off' | 'on'  
**Default:** 'off'

**Output as nonvirtual bus — Specify virtual or nonvirtual bus output**

off (default) | on
Specify whether the output for a top-level Inport block used to load bus data is virtual or nonvirtual.

- Select this parameter to specify a nonvirtual bus output.
- Clear this parameter to specify a virtual bus output.

**Tips**

- All signals in a nonvirtual bus must have the same sample time, even if the associated bus object specifies inherited sample time for some elements. Any operation that would result in a nonvirtual bus containing signals with different sample rates generates an error. You cannot load multirate data for a nonvirtual bus. See “Multirate Bus Input for Referenced Models” for details on how to pass signals with different sample rates into a referenced model as a nonvirtual bus.

To load multirate data for a bus, clear the **Output as nonvirtual bus** parameter, and set the **Sample time** parameter to inherited (-1).

- For the top model in a model reference hierarchy, code generation creates a C structure to represent the nonvirtual bus output.
- For referenced models, select this option to create a C structure in generated code. Otherwise, code generation creates an argument for each leaf element of the bus.

**Dependency**

This parameter is only available for top-level Inport blocks with **Bus: <object name>** selected for the **Data type** parameter.

**Programmatic Use**

**Block Parameter:** BusOutputAsStruct  
**Type:** character vector  
**Values:** 'off' | 'on'  
**Default:** 'off'

**Unit (e.g., m, m/s^2, N*m) — Physical unit of the input signal to the block**

Specify the physical unit of the input signal to the block. To specify a unit, begin typing in the text box. As you type, the parameter displays potential matching units. For a list of supported units, see Allowed Unit Systems.

To constrain the unit system, click the link to the right of the parameter:
• If a Unit System Configuration block exists in the component, its dialog box opens. Use that dialog box to specify allowed and disallowed unit systems for the component.

• If a Unit System Configuration block does not exist in the component, the model Configuration Parameters dialog box displays. Use that dialog box to specify allowed and disallowed unit systems for the model.

**Programmatic Use**

**Block Parameter:** Unit  
**Type:** character vector  
**Values:** 'inherit' | '<Enter unit>'  
**Default:** 'inherit'

**Port dimensions (-1 for inherited) — Port dimensions**  
-1 (default) | integer | [integer integer]

Specify the dimensions of the output signal for this Inport block.

<table>
<thead>
<tr>
<th>-1</th>
<th>The port can load data for a signal with any dimensions. The port inherits dimensions from the connected signal.</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>The port can load data for a signal that is a vector of size N.</td>
</tr>
<tr>
<td>[R C]</td>
<td>The port can load data for a matrix signal having R rows and C columns.</td>
</tr>
</tbody>
</table>

**Programmatic Use**

**Block Parameter:** PortDimensions  
**Type:** character vector  
**Values:** '-1' | integer | [integer integer]  
**Default:** '-1'

**Variable-size signal — Allow variable-size signals**  
Inherit (default) | No | Yes

Specify the type of signals allowed out of this port. To allow variable-size and fixed-size signals, select Inherit. To allow only variable-size signals, select Yes. To allow only fixed-size signals, select No.

**Dependencies**

When the signal at this port is a variable-size signal, the **Port dimensions** parameter specifies the maximum dimensions of the signal.
Command-Line Information
Parameter: VarSizeSig
Type: character vector
Value: 'Inherit' | 'No' | 'Yes'
Default: 'Inherit'

Sample time (-1 for inherited) — Specify sample time
-1 (default) | scalar

Specify the discrete interval between sample time hits or specify another appropriate sample time such as continuous or inherited.

By default, the block inherits its sample time based upon the context of the block within the model. To set a different sample time, enter a valid sample time based upon the table in “Types of Sample Time”.

Programmatic Use
Block Parameter: SampleTime
Type: character vector
Values: scalar
Default: '-1'

Signal type — Output signal type
auto (default) | real | complex

Specify the numeric type of the signal output. To choose the numeric type of the signal that is connected to its input, select auto. Otherwise, choose a real or complex signal type.

Programmatic Use
Block Parameter: SignalType
Type: character vector
Values: 'auto' | 'real' | 'complex'
Default: 'auto'

Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>bus</th>
<th>double</th>
<th>enumerated</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
<th>string</th>
</tr>
</thead>
</table>

1-1009
### Extended Capabilities

#### C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

#### HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

**HDL Architecture**

This block has a single, default HDL architecture.

<table>
<thead>
<tr>
<th>Direct Feedthrough</th>
<th>no</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multidimensional Signals</td>
<td>yes</td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>yes</td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
</tr>
</tbody>
</table>
**HDL Block Properties**

<table>
<thead>
<tr>
<th>General</th>
<th>BidirectionalPort Setting</th>
<th>Description</th>
</tr>
</thead>
</table>
|         | on                        | Specify the port as bidirectional. The following requirements apply:  
|         |                           | • The port must be in a Subsystem block with black box implementation.  
|         |                           | • There must also be no logic between the bidirectional port and the corresponding top-level DUT subsystem port.  
|         | off (default)             | Do not specify the port as bidirectional.  
|         |                           | For more information, see “Specify Bidirectional Ports” (HDL Coder). |
Target Specification

| IOInterface | Target platform interface type for DUT ports, specified as a character vector. The IOInterface block property is ignored for Inport and Outport blocks that are not DUT ports.

To specify valid IOInterface settings, use the HDL Workflow Advisor:

1. In the HDL Workflow Advisor, in the Set Target > Set Target Interface step, in the Target platform interface table, in the Target Platform Interfaces column, use the drop-down list to set the target platform interface type.

2. Save the model.

The IOInterface value is saved as an HDL block property of the port.

For example, to view the IOInterface value, if the full path to your DUT port is hdlcoder_led_blinking/led_counter/LED, enter:

```matlab
hdlink('hdlcoder_led_blinking/led_counter/LED', 'IOInterface')
```
**Target Specification**

<table>
<thead>
<tr>
<th>IOInterfaceMapping</th>
<th>Target platform interface port mapping for DUT ports, specified as a character vector. The IOInterfaceMapping block property is ignored for Inport and Outport blocks that are not DUT ports.</th>
</tr>
</thead>
</table>

To specify valid IOInterfaceMapping settings, use the HDL Workflow Advisor:

1. In the HDL Workflow Advisor, in the **Set Target > Set Target Interface** step, in the **Target platform interface table**, in the **Target Platform Interfaces** column, use the drop-down list to set the target platform interface type.

2. In the **Bit Range / Address / FPGA Pin** column, if you want to change the default value, enter a target platform interface mapping.

3. Save the model.

The IOInterfaceMapping value is saved as an HDL block property of the port.

For example, to view the IOInterfaceMapping value, if the full path to your DUT port is `hdlcoder_led_blinking/led_counter/LED`, enter:

```matlab
hdlget_param('hdlcoder_led_blinking/led_counter/LED','...','IOInterfaceMapping')
```

---

**PLC Code Generation**

Generate Structured Text code using Simulink® PLC Coder™.

**Fixed-Point Conversion**

Design and simulate fixed-point systems using Fixed-Point Designer™.

**See Also**

Asynchronous Task Specification | In Bus Element | Outport
Topics
“Load Data to Root-Level Input Ports”
“Using Function-Call Subsystems”
“Map Data Using Root Inport Mapper Tool”
“Load Big Data for Simulations”
“Comparison of Signal Loading Techniques”
“Map Root Inport Signal Data”

Introduced before R2006a
Integrator

Integrate signal

Library:
- Simulink / Commonly Used Blocks
- Simulink / Continuous

Description

The Integrator block outputs the value of the integral of its input signal with respect to time.

Simulink treats the Integrator block as a dynamic system with one state. The block dynamics are given by:

\[
\begin{align*}
\dot{x}(t) &= u(t) \\
y(t) &= x(t) \\
x(t_0) &= x_0
\end{align*}
\]

where:
- \(u\) is the block input.
- \(y\) is the block output.
- \(x\) is the block state.
- \(x_0\) is the initial condition of \(x\).

While these equations define an exact relationship in continuous time, Simulink uses numerical approximation methods to evaluate them with finite precision. Simulink can use several different numerical integration methods to compute the output of the block, each with advantages in particular applications. Use the **Solver** pane of the Configuration Parameters dialog box (see “Solver Pane”) to select the technique best suited to your application.

The selected solver computes the output of the Integrator block at the current time step, using the current input value and the value of the state at the previous time step. To support this computational model, the Integrator block saves its output at the current
time step for use by the solver to compute its output at the next time step. The block also provides the solver with an initial condition for use in computing the block's initial state at the beginning of a simulation. The default value of the initial condition is 0. Use the block parameter dialog box to specify another value for the initial condition or create an initial value input port on the block.

Use the parameter dialog box to:

• Define upper and lower limits on the integral
• Create an input that resets the block's output (state) to its initial value, depending on how the input changes
• Create an optional state output so that the value of the block's output can trigger a block reset

Use the Discrete-Time Integrator block to create a purely discrete system.

**Defining Initial Conditions**

You can define the initial conditions as a parameter on the block dialog box or input them from an external signal:

• To define the initial conditions as a block parameter, specify the **Initial condition source** parameter as **internal** and enter the value in the **Initial condition** field.
• To provide the initial conditions from an external source, specify the **Initial condition source** parameter as **external**. An additional input port appears under the block input.

![Diagram of Discrete-Time Integrator block]

**Note** If the integrator limits its output (see “Limiting the Integral” on page 1-1017), the initial condition must fall inside the integrator's saturation limits. If the initial condition is outside the block saturation limits, the block displays an error message.
**Limiting the Integral**

To prevent the output from exceeding specifiable levels, select the **Limit output** check box and enter the limits in the appropriate parameter fields. This action causes the block to function as a limited integrator. When the output reaches the limits, the integral action is turned off to prevent integral wind up. During a simulation, you can change the limits but you cannot change whether the output is limited. The block determines output as follows:

- When the integral is less than or equal to the **Lower saturation limit**, the output is held at the **Lower saturation limit**.
- When the integral is between the **Lower saturation limit** and the **Upper saturation limit**, the output is the integral.
- When the integral is greater than or equal to the **Upper saturation limit**, the output is held at the **Upper saturation limit**.

To generate a signal that indicates when the state is being limited, select the **Show saturation port** check box. A saturation port appears below the block output port.

The signal has one of three values:

- 1 indicates that the upper limit is being applied.
- 0 indicates that the integral is not limited.
- –1 indicates that the lower limit is being applied.

When you select this check box, the block has three zero crossings: one to detect when it enters the upper saturation limit, one to detect when it enters the lower saturation limit, and one to detect when it leaves saturation.

**Note** For the Integrator Limited block, by default, **Limit output** is selected, **Upper saturation limit** is set to 1, and **Lower saturation limit** is set to 0.
Wrapping Cyclic States

Several physical phenomena are cyclic, periodic, or rotary in nature. Objects or machinery that exhibit rotational movement and oscillators are examples of such phenomena.

Modeling these phenomena in Simulink involves integrating the rate of change of the periodic or cyclic signals to obtain the state of the movement.

The drawback with this approach, however, is that over long simulation time spans, the states representing periodic or cyclic signals integrate to large values. Further, computing the sine or cosine of these signals takes an increasingly large amount of time because of angle reduction. The large signals values also negatively impact solver performance and accuracy.

One approach for overcoming this drawback is to reset the angular state to 0 when it reaches 2π (or to −π when it reaches π, for numerical symmetry). This approach improves the accuracy of sine and cosine computations and reduces angle reduction time. But it also requires zero-crossing detection and introduces solver resets, which slow down the simulation for variable step solvers, particularly in large models.

To eliminate solver resets at wrap points, the Integrator block supports wrapped states that you can enable by checking 
Wrap state
on the block parameter dialog box. When you enable 
Wrap state, the block icon changes to indicate that the block has wrapping states.

Simulink allows wrapping states that are bounded by upper and lower values parameters of the wrapped state. The algorithm for determining wrapping states is given by:

\[
y = \begin{cases} 
  x & x \in [x_l, x_u) \\
  x - (x_u - x_l) & \frac{x - x_l}{x_u - x_l} \end{cases}
\]

otherwise

1-1018
where:

- $x_l$ is the lower value of the wrapped state.
- $x_u$ is the upper value of the wrapped state.
- $y$ is the output.

The support for wrapping states provides these advantages.

- It eliminates simulation instability when your model approaches large angles and large state values.
- It reduces the number of solver resets during simulation and eliminates the need for zero-crossing detection, improving simulation time.
- It eliminates large angle values, speeding up computation of trigonometric functions on angular states.
- It improves solver accuracy and performance and enables unlimited simulation time.

**Resetting the State**

The block can reset its state to the specified initial condition based on an external signal. To cause the block to reset its state, select one of the **External reset** choices. A trigger port appears below the block's input port and indicates the trigger type.

- Select **rising** to reset the state when the reset signal rises from a negative or zero value to a positive value.
- Select **falling** to reset the state when the reset signal falls from a positive value to a zero or negative value.
- Select **either** to reset the state when the reset signal changes from zero to a nonzero value, from a nonzero value to zero, or changes sign.
- Select **level** to reset the state when the reset signal is nonzero at the current time step or changes from nonzero at the previous time step to zero at the current time step.
- Select **level hold** to reset the state when the reset signal is nonzero at the current time step.
The reset port has direct feedthrough. If the block output feeds back into this port, either directly or through a series of blocks with direct feedthrough, an algebraic loop results (see “Algebraic Loop Concepts”). Use the Integrator block's state port to feed back the block's output without creating an algebraic loop.

**Note** To be compliant with the Motor Industry Software Reliability Association (MISRA) software standard, your model must use Boolean signals to drive the external reset ports of Integrator blocks.

**About the State Port**

Selecting the **Show state port** check box on the Integrator block's parameter dialog box causes an additional output port, the state port, to appear at the top of the Integrator block.

The output of the state port is the same as the output of the block's standard output port except for the following case. If the block is reset in the current time step, the output of the state port is the value that would have appeared at the block's standard output if the block had not been reset. The state port's output appears earlier in the time step than the output of the Integrator block's output port. Use the state port to avoid creating algebraic loops in these modeling scenarios:

- Self-resetting integrators (see “Creating Self-Resetting Integrators” on page 1-1021)
- Handing off a state from one enabled subsystem to another (see “Handing Off States Between Enabled Subsystems” on page 1-1022)

**Note** When updating a model, Simulink checks that the state port applies to one of these two scenarios. If not, an error message appears. Also, you cannot log the output of this port in a referenced model that executes in Accelerator mode. If logging is enabled for the port, Simulink generates a "signal not found" warning during execution of the referenced model.
Creating Self-Resetting Integrators

The Integrator block's state port helps you avoid an algebraic loop when creating an integrator that resets itself based on the value of its output. Consider, for example, the following model.

![Model diagram](image)

This model tries to create a self-resetting integrator by feeding the integrator's output, subtracted from 1, back into the integrator's reset port. However, the model creates an algebraic loop. To compute the integrator block's output, Simulink software needs to know the value of the block's reset signal, and vice versa. Because the two values are mutually dependent, Simulink software cannot determine either. Therefore, an error message appears if you try to simulate or update this model.

The following model uses the integrator's state port to avoid the algebraic loop.

![Model diagram](image)

In this version, the value of the reset signal depends on the value of the state port. The value of the state port is available earlier in the current time step than the value of the
integrator block's output port. Therefore, Simulink can determine whether the block needs to be reset before computing the block's output, thereby avoiding the algebraic loop.

**Handing Off States Between Enabled Subsystems**

The state port helps you avoid an algebraic loop when passing a state between two enabled subsystems. Consider, for example, the following model.

The enabled subsystems, A and B, contain the following blocks:
In this model, a constant input signal drives two enabled subsystems that integrate the signal. A pulse generator generates an enabling signal that causes execution to alternate between the two subsystems. The enable port of each subsystem is set to reset, which causes the subsystem to reset its integrator when it becomes active. Resetting the integrator causes the integrator to read the value of its initial condition port. The initial condition port of the integrator in each subsystem is connected to the output port of the integrator in the other subsystem.

This connection is intended to enable continuous integration of the input signal as execution alternates between two subsystems. However, the connection creates an algebraic loop. To compute the output of A, Simulink needs to know the output of B, and vice versa. Because the outputs are mutually dependent, Simulink cannot compute the output values. Therefore, an error message appears if you try to simulate or update this model.

The following version of the same model uses the integrator state port to avoid creating an algebraic loop when handing off the state.

The enabled subsystems, A and B, contain the following blocks:
In this model, the initial condition of the integrator in A depends on the value of the state port of the integrator in B, and vice versa. The values of the state ports are updated earlier in the simulation time step than the values of the integrator output ports. Therefore, Simulink can compute the initial condition of either integrator without knowing the final output value of the other integrator. For another example of using the state port to hand off states between conditionally executed subsystems, see the sldemo_clutch model.

### Specifying the Absolute Tolerance for the Block Outputs

By default Simulink software uses the absolute tolerance value specified in the Configuration Parameters dialog box (see “Error Tolerances for Variable-Step Solvers”) to compute the output of the Integrator block. If this value does not provide sufficient error control, specify a more appropriate value in the Absolute tolerance field of the Integrator block dialog box. The value that you specify is used to compute all the block outputs.

### Selecting All Options

When you select all options, the block icon looks like this.
Ports

The Integrator block accepts and outputs signals of type `double` on its data ports. The external reset port accepts signals of type `double` or `Boolean`.

Input

**Port_1 — Integrand signal**
real scalar or array

Signal that needs to be integrated.

Data Types: `double`

**External Reset — Reset state to initial conditions**
real scalar or array

Reset the state to the specified initial conditions based on an external signal. See “Resetting the State” on page 1-1019.

Dependencies

To enable this port, enable the **External Reset** parameter.

Data Types: `Boolean`

**$x_0$ — Initial condition**
real scalar or array

Set the initial condition of the block's state from an external signal.

Dependencies

To enable this port, set the **Initial Conditions** parameter to `external`.

Data Types: `double`

Output

**Port_1 — Output signal**
real scalar or array

Output the integrated state.
Data Types: double

**Port 2 — Show output saturation**

-1 | 0 | 1

Indicate when the state is being limited. The signal has a value of 1 when the integral is limited by the specified **Upper saturation limit**. When the signal is limited by the **Lower saturation limit**, the signal value is -1. When the integral is between the saturation limits, the signal value is 0. See “Limiting the Integral” on page 1-1017.

Data Types: double

**Port 3 — State**

real scalar or array

Output the state of the block. See “About the State Port” on page 1-1020.

**Dependencies**

Enable this port by enabling the **Show state port** parameter.

Data Types: double

**Parameters**

**External reset — Reset states to their initial conditions**

none (default) | rising | falling | either | level | level hold

Specify the type of trigger to use for the external reset signal.

- Select **rising** to reset the state when the reset signal rises from a negative or zero value to a positive value.
- Select **falling** to reset the state when the reset signal falls from a positive value to a zero or negative value.
- Select **either** to reset the state when the reset signal changes from zero to a nonzero value, from a nonzero value to zero, or changes sign.
- Select **level** to reset the state when the reset signal is nonzero at the current time step or changes from nonzero at the previous time step to zero at the current time step.
- Select **level hold** to reset the state when the reset signal is nonzero at the current time step.
**Programmatic Use**

**Block Parameter:** `ExternalReset`  
**Type:** character vector, string  
**Values:**  
- `'none'`  
- `'rising'`  
- `'falling'`  
- `'either'`  
- `'level'`  
- `'level hold'`  
**Default:** `'none'`

**Initial condition source — Select source of initial condition**  
`internal` (default) | `external`

Select source of initial condition:

- **internal** — Get the initial conditions of the states from the **Initial condition** block parameter.
- **external** — Get the initial conditions of the states from an external block, via the **IC** input port.

**Dependencies**

Selecting `internal` enables the **Initial condition** parameter.

Selecting `external` disables the **Initial condition** parameter and enables the **IC** input port.

**Programmatic Use**

**Block Parameter:** `InitialConditionSource`  
**Type:** character vector, string  
**Values:**  
- `'internal'`  
- `'external'`  
**Default:** `'internal'`

**Initial condition — Initial state**  
`0` (default) | real scalar or array

Set the initial state of the Integrator block.

**Tips**

Simulink software does not allow the initial condition of this block to be `inf` or `NaN`.

**Dependencies**

Setting **Initial condition source** to `internal` enables this parameter.

Setting **Initial condition source** to `external` disables this parameter.
**Programmatic Use**

**Block Parameter:** InitialCondition  
**Type:** scalar or vector  
**Default:** '0'

**Limit output — Limit block output values to specified range**  
off (default for Integrator) | on (default for Integrator Limited)

Limit the block's output to a value between the **Lower saturation limit** and **Upper saturation limit** parameters.

- Selecting this check box limits the block output to a value between the **Lower saturation limit** and **Upper saturation limit** parameters.
- Clearing this check box does not limit the block output values.

**Dependencies**

Selecting this parameter enables the **Lower saturation limit** and **Upper saturation limit** parameters.

**Programmatic Use**

**Block Parameter:** LimitOutput  
**Type:** character vector, string  
**Values:** 'off' | 'on'  
**Default:** 'off'

**Upper saturation limit — Upper limit for the integral**  
inf (default) | scalar | vector | matrix

Specify the upper limit for the integral as a scalar, vector, or matrix. You must specify a value between the **Output minimum** and **Output maximum** parameter values.

**Dependencies**

To enable this parameter, select the **Limit output** check box.

**Programmatic Use**

**Block Parameter:** UpperSaturationLimit  
**Type:** character vector, string  
**Values:** scalar | vector | matrix  
**Default:** 'inf'
**Lower saturation limit — Lower limit for the integral**
-\( -\infty \) (default) | scalar | vector | matrix

Specify the lower limit for the integral as a scalar, vector, or matrix. You must specify a value between the **Output minimum** and **Output maximum** parameter values.

**Dependencies**

To enable this parameter, select the **Limit output** check box.

**Programmatic Use**

**Block Parameter:** LowerSaturationLimit
**Type:** character vector, string
**Values:** scalar | vector | matrix
**Default:** ‘-\( \infty \)’

**Wrap state — Enable wrapping of states**

**off** (default) | **on**

Enable wrapping of states between the **Wrapped state upper value** and **Wrapped state lower value** parameters. Enabling wrap states eliminates the need for zero-crossing detection, reduces solver resets, improves solver performance and accuracy, and increases simulation time span when modeling rotary and cyclic state trajectories.

If you specify **Wrapped state upper value** as \( \infty \) and **Wrapped state lower value** as \(-\infty\), wrapping does not occur.

**Dependencies**

Selecting this parameter enables **Wrapped state upper value** and **Wrapped state lower value** parameters.

**Programmatic Use**

**Block Parameter:** WrapState
**Type:** character vector, string
**Values:** 'off' | 'on'
**Default:** 'off'

**Wrapped state upper value — Specify upper value for wrapped state**

\( \pi \) (default) | scalar or vector

Upper limit of the block output.
Dependencies

Selecting Wrap state enables this parameter.

Programmatic Use

Block Parameter: WrappedStateUpperValue
Type: scalar or vector
Values: '2*pi'
Default: 'pi'

Wrapped state lower value — Specify lower value for wrap state
-π (default) | scalar or vector

Lower limit of the block output.

Dependencies

Selecting Wrap state enables this parameter.

Programmatic Use

Block Parameter: WrappedStateLowerValue
Type: scalar or vector
Values: '0'
Default: '-π'

Show saturation port — Enable saturation output port
off (default) | on

Select this check box to add a saturation output port to the block. When you clear this check box, the block does not have a saturation output port.

Dependencies

Selecting this parameter enables a saturation output port.

Programmatic Use

Block Parameter: ShowSaturationPort
Type: character vector, string
Values: 'off' | 'on'
Default: 'off'

Show state port — Enable state output port
off (default) | on
Select this check box to add a state output port to the block. When you clear this check box, the block does not have a state output port.

**Dependencies**

Selecting this parameter enables a state output port.

**Programmatic Use**

**Block Parameter:** ShowStatePort  
**Type:** character vector, string  
**Values:** 'off' | 'on'  
**Default:** 'off'

**Absolute tolerance — Absolute tolerance for block states**

- **auto** (default) | real scalar or vector

- If you enter **auto** or -1, then Simulink uses the absolute tolerance value in the Configuration Parameters dialog box (see “Solver Pane”) to compute block states.
- If you enter a real scalar, then that value overrides the absolute tolerance in the Configuration Parameters dialog box for computing all block states.
- If you enter a real vector, then the dimension of that vector must match the dimension of the continuous states in the block. These values override the absolute tolerance in the Configuration Parameters dialog box.

**Programmatic Use**

**Block Parameter:** AbsoluteTolerance  
**Type:** character vector, string, scalar, or vector  
**Values:** 'auto' | '-1' | any positive real scalar or vector  
**Default:** 'auto'

**Ignore limit and reset when linearizing — Treat block as unresettable and output unlimited**

- **off** (default) | **on**

Cause Simulink linearization commands to treat this block as unresettable and as having no limits on its output, regardless of the settings of the reset and output limitation options of the block.

**Tip**

Use this check box to linearize a model around an operating point that causes the integrator to reset or saturate.
Programmatic Use

**Block Parameter:** IgnoreLimit
Type: character vector, string
Values: 'off' | 'on'
Default: 'off'

**Enable zero-crossing detection — Enable zero-crossing detection**
on (default) | Boolean

Select to enable zero-crossing detection. For more information, see “Zero-Crossing Detection” in the Simulink documentation.

Programmatic Use
**Block Parameter:** ZeroCross
Type: character vector, string
Values: 'off' | 'on'
Default: 'on'

**State Name (e.g., 'position') — Assign unique name to each state**
' ' (default) | character vector | string

- To assign a name to a single state, enter the name between quotes, for example, 'velocity'.
- To assign names to multiple states, enter a comma-delimited list surrounded by braces, for example, {'a', 'b', 'c'}. Each name must be unique.
- The state names apply only to the selected block.
- The number of states must divide evenly among the number of state names.
- You can specify fewer names than states, but you cannot specify more names than states.

For example, you can specify two names in a system with four states. The first name applies to the first two states and the second name to the last two states.

- To assign state names with a variable in the MATLAB workspace, enter the variable without quotes. A variable can be a character vector, string, cell array, or structure.

Programmatic Use
**Block Parameter:** ContinuousStateAttributes
Type: character vector, string
Values: ' ' | user-defined
Default: ' '  

**Block Characteristics**

<table>
<thead>
<tr>
<th>Data Types</th>
<th>double</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>yes</td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>no</td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>yes</td>
</tr>
</tbody>
</table>

**Extended Capabilities**

**C/C++ Code Generation**
Generate C and C++ code using Simulink® Coder™.

- Consider discretizing the model
- Not recommended for production code

**See Also**
Discrete-Time Integrator | Second-Order Integrator

**Introduced before R2006a**
Integrator Limited

Integrate signal
Library: Simulink / Continuous

Description

The Integrator Limited block is identical to the Integrator block with the exception that the output of the block is limited based on the upper and lower saturation limits. See “Limiting the Integral” on page 1-1037 for details.

Simulink treats the Integrator block as a dynamic system with one state. The block dynamics are given by:

\[
\begin{align*}
\dot{x}(t) &= u(t) \\
y(t) &= x(t) \\
x(t_0) &= x_0
\end{align*}
\]

where:

- \( u \) is the block input.
- \( y \) is the block output.
- \( x \) is the block state.
- \( x_0 \) is the initial condition of \( x \).

While these equations define an exact relationship in continuous time, Simulink uses numerical approximation methods to evaluate them with finite precision. Simulink can use several different numerical integration methods to compute the output of the block, each with advantages in particular applications. Use the Solver pane of the Configuration Parameters dialog box (see “Solver Pane”) to select the technique best suited to your application.

The selected solver computes the output of the Integrator block at the current time step, using the current input value and the value of the state at the previous time step. To
support this computational model, the Integrator block saves its output at the current time step for use by the solver to compute its output at the next time step. The block also provides the solver with an initial condition for use in computing the block's initial state at the beginning of a simulation. The default value of the initial condition is 0. Use the block parameter dialog box to specify another value for the initial condition or create an initial value input port on the block.

Use the parameter dialog box to:

• Define upper and lower limits on the integral
• Create an input that resets the block's output (state) to its initial value, depending on how the input changes
• Create an optional state output so that the value of the block's output can trigger a block reset

Use the Discrete-Time Integrator block to create a purely discrete system.

**Defining Initial Conditions**

You can define the initial conditions as a parameter on the block dialog box or input them from an external signal:

• To define the initial conditions as a block parameter, specify the **Initial condition source** parameter as **internal** and enter the value in the **Initial condition** field.
• To provide the initial conditions from an external source, specify the **Initial condition source** parameter as **external**. An additional input port appears under the block input.

**Note** If the integrator limits its output (see “Limiting the Integral” on page 1-1037), the initial condition must fall inside the integrator's saturation limits. If the initial condition is outside the block saturation limits, the block displays an error message.
Wrapping Cyclic States

Several physical phenomena are cyclic, periodic, or rotary in nature. Objects or machinery that exhibit rotational movement and oscillators are examples of such phenomena.

Modeling these phenomena in Simulink involves integrating the rate of change of the periodic or cyclic signals to obtain the state of the movement.

The drawback with this approach, however, is that over long simulation time spans, the states representing periodic or cyclic signals integrate to large values. Further, computing the sine or cosine of these signals takes an increasingly large amount of time because of angle reduction. The large signals values also negatively impact solver performance and accuracy.

One approach for overcoming this drawback is to reset the angular state to 0 when it reaches 2π (or to –π when it reaches π, for numerical symmetry). This approach improves the accuracy of sine and cosine computations and reduces angle reduction time. But it also requires zero-crossing detection and introduces solver resets, which slow down the simulation for variable step solvers, particularly in large models.

To eliminate solver resets at wrap points, the Integrator block supports wrapped states that you can enable by checking **Wrap state** on the block parameter dialog box. When you enable **Wrap state**, the block icon changes to indicate that the block has wrapping states.

Simulink allows wrapping states that are bounded by upper and lower values parameters of the wrapped state. The algorithm for determining wrapping states is given by:

\[
y = \begin{cases} 
  x & x \in [x_l, x_u) \\
  x - \frac{(x_u - x_l)(x - x_l)}{|x_u - x_l|} & \text{otherwise}
\end{cases}
\]
where:
- $x_l$ is the lower value of the wrapped state.
- $x_u$ is the upper value of the wrapped state.
- $y$ is the output.

The support for wrapping states provides these advantages.
- It eliminates simulation instability when your model approaches large angles and large state values.
- It reduces the number of solver resets during simulation and eliminates the need for zero-crossing detection, improving simulation time.
- It eliminates large angle values, speeding up computation of trigonometric functions on angular states.
- It improves solver accuracy and performance and enables unlimited simulation time.

**Limiting the Integral**

To prevent the output from exceeding specifiable levels, select the **Limit output** check box and enter the limits in the appropriate parameter fields. This action causes the block to function as a limited integrator. When the output reaches the limits, the integral action is turned off to prevent integral wind up. During a simulation, you can change the limits but you cannot change whether the output is limited. The block determines output as follows:

- When the integral is less than or equal to the **Lower saturation limit**, the output is held at the **Lower saturation limit**.
- When the integral is between the **Lower saturation limit** and the **Upper saturation limit**, the output is the integral.
- When the integral is greater than or equal to the **Upper saturation limit**, the output is held at the **Upper saturation limit**.

To generate a signal that indicates when the state is being limited, select the **Show saturation port** check box. A saturation port appears below the block output port.
The signal has one of three values:

- 1 indicates that the upper limit is being applied.
- 0 indicates that the integral is not limited.
- -1 indicates that the lower limit is being applied.

When you select this check box, the block has three zero crossings: one to detect when it enters the upper saturation limit, one to detect when it enters the lower saturation limit, and one to detect when it leaves saturation.

**Note** For the Integrator Limited block, by default, Limit output is selected, Upper saturation limit is set to 1, and Lower saturation limit is set to 0.

**Resetting the State**

The block can reset its state to the specified initial condition based on an external signal. To cause the block to reset its state, select one of the External reset choices. A trigger port appears below the block's input port and indicates the trigger type.

- Select rising to reset the state when the reset signal rises from a negative or zero value to a positive value.
- Select falling to reset the state when the reset signal falls from a positive value to a zero or negative value.
- Select either to reset the state when the reset signal changes from zero to a nonzero value, from a nonzero value to zero, or changes sign.
- Select level to reset the state when the reset signal is nonzero at the current time step or changes from nonzero at the previous time step to zero at the current time step.
- Select level hold to reset the state when the reset signal is nonzero at the current time step.

The reset port has direct feedthrough. If the block output feeds back into this port, either directly or through a series of blocks with direct feedthrough, an algebraic loop results.
(see “Algebraic Loop Concepts”). Use the Integrator block's state port to feed back the block's output without creating an algebraic loop.

**Note** To be compliant with the Motor Industry Software Reliability Association (MISRA) software standard, your model must use Boolean signals to drive the external reset ports of Integrator blocks.

### About the State Port

Selecting the **Show state port** check box on the Integrator block's parameter dialog box causes an additional output port, the state port, to appear at the top of the Integrator block.

The output of the state port is the same as the output of the block's standard output port except for the following case. If the block is reset in the current time step, the output of the state port is the value that would have appeared at the block's standard output if the block had not been reset. The state port's output appears earlier in the time step than the output of the Integrator block's output port. Use the state port to avoid creating algebraic loops in these modeling scenarios:

- Self-resetting integrators (see “Creating Self-Resetting Integrators” on page 1-1040)
- Handing off a state from one enabled subsystem to another (see “Handing Off States Between Enabled Subsystems” on page 1-1041)

**Note** When updating a model, Simulink checks that the state port applies to one of these two scenarios. If not, an error message appears. Also, you cannot log the output of this port in a referenced model that executes in Accelerator mode. If logging is enabled for the port, Simulink generates a "signal not found" warning during execution of the referenced model.
Creating Self-Resetting Integrators

The Integrator block's state port helps you avoid an algebraic loop when creating an integrator that resets itself based on the value of its output. Consider, for example, the following model.

This model tries to create a self-resetting integrator by feeding the integrator's output, subtracted from 1, back into the integrator's reset port. However, the model creates an algebraic loop. To compute the integrator block's output, Simulink software needs to know the value of the block's reset signal, and vice versa. Because the two values are mutually dependent, Simulink software cannot determine either. Therefore, an error message appears if you try to simulate or update this model.

The following model uses the integrator's state port to avoid the algebraic loop.

In this version, the value of the reset signal depends on the value of the state port. The value of the state port is available earlier in the current time step than the value of the
integrator block's output port. Therefore, Simulink can determine whether the block
needs to be reset before computing the block's output, thereby avoiding the algebraic
loop.

**Handing Off States Between Enabled Subsystems**

The state port helps you avoid an algebraic loop when passing a state between two
enabled subsystems. Consider, for example, the following model.

The enabled subsystems, A and B, contain the following blocks:
In this model, a constant input signal drives two enabled subsystems that integrate the signal. A pulse generator generates an enabling signal that causes execution to alternate between the two subsystems. The enable port of each subsystem is set to reset, which causes the subsystem to reset its integrator when it becomes active. Resetting the integrator causes the integrator to read the value of its initial condition port. The initial condition port of the integrator in each subsystem is connected to the output port of the integrator in the other subsystem.

This connection is intended to enable continuous integration of the input signal as execution alternates between two subsystems. However, the connection creates an algebraic loop. To compute the output of A, Simulink needs to know the output of B, and vice versa. Because the outputs are mutually dependent, Simulink cannot compute the output values. Therefore, an error message appears if you try to simulate or update this model.

The following version of the same model uses the integrator state port to avoid creating an algebraic loop when handing off the state.

The enabled subsystems, A and B, contain the following blocks:
In this model, the initial condition of the integrator in A depends on the value of the state port of the integrator in B, and vice versa. The values of the state ports are updated earlier in the simulation time step than the values of the integrator output ports. Therefore, Simulink can compute the initial condition of either integrator without knowing the final output value of the other integrator. For another example of using the state port to hand off states between conditionally executed subsystems, see the sldemo_clutch model.

### Specifying the Absolute Tolerance for the Block Outputs

By default Simulink software uses the absolute tolerance value specified in the Configuration Parameters dialog box (see “Error Tolerances for Variable-Step Solvers”) to compute the output of the Integrator block. If this value does not provide sufficient error control, specify a more appropriate value in the **Absolute tolerance** field of the Integrator block dialog box. The value that you specify is used to compute all the block outputs.

### Selecting All Options

When you select all options, the block icon looks like this.
Ports

Input

Port_1 — Integrand signal  
real scalar or array

Signal that needs to be integrated.  
Data Types: double

External Reset — Reset state to initial conditions  
real scalar or array

Reset the state to the specified initial conditions based on an external signal. See “Resetting the State” on page 1-1038.  
 Dependencies

To enable this port, enable the External Reset parameter.  
Data Types: Boolean

x_0 — Initial condition  
real scalar or array

Set the initial condition of the block's state from an external signal.  
 Dependencies

To enable this port, set the Initial Conditions parameter to external.  
Data Types: double

Output

Port_1 — Output signal  
real scalar or array

Output the integrated state.  
Data Types: double
**Port_2 — Show output saturation**

| -1 | 0 | 1 |

Indicate when the state is being limited. The signal has a value of 1 when the integral is limited by the specified **Upper saturation limit**. When the signal is limited by the **Lower saturation limit**, the signal value is -1. When the integral is between the saturation limits, the signal value is 0. See “Limiting the Integral” on page 1-1037.

Data Types: double

**Port_3 — State**
real scalar or array

Output the state of the block. See “About the State Port” on page 1-1039.

**Dependencies**
Enable this port by enabling the **Show state port** parameter.

Data Types: double

**Parameters**

**External reset — Reset states to their initial conditions**

none (default) | rising | falling | either | level | level hold

Specify the type of trigger to use for the external reset signal.

- Select **rising** to reset the state when the reset signal rises from a negative or zero value to a positive value.
- Select **falling** to reset the state when the reset signal falls from a positive value to a zero or negative value.
- Select **either** to reset the state when the reset signal changes from zero to a nonzero value, from a nonzero value to zero, or changes sign.
- Select **level** to reset the state when the reset signal is nonzero at the current time step or changes from nonzero at the previous time step to zero at the current time step.
- Select **level hold** to reset the state when the reset signal is nonzero at the current time step.
**Programmatic Use**

**Block Parameter:** ExternalReset  
**Type:** character vector, string  
**Values:** 'none' | 'rising' | 'falling' | 'either' | 'level' | 'level hold'  
**Default:** 'none'

**Initial condition source — Select source of initial condition**  
internal (default) | external

Select source of initial condition:

- **internal** — Get the initial conditions of the states from the Initial condition block parameter.
- **external** — Get the initial conditions of the states from an external block, via the IC input port.

**Dependencies**

Selecting **internal** enables the **Initial condition** parameter.

Selecting **external** disables the **Initial condition** parameter and enables the IC input port.

**Programmatic Use**

**Block Parameter:** InitialConditionSource  
**Type:** character vector, string  
**Values:** 'internal' | 'external'  
**Default:** 'internal'

**Initial condition — Initial state**  
0 (default) | real scalar or array

Set the initial state of the Integrator block.

**Tips**

Simulink software does not allow the initial condition of this block to be inf or NaN.

**Dependencies**

Setting **Initial condition source** to internal enables this parameter.

Setting **Initial condition source** to external disables this parameter.
Programmatic Use

Block Parameter: InitialCondition
Type: scalar or vector
Default: '0'

Limit output — Limit block output values to specified range
off (default for Integrator) | on (default for Integrator Limited)

Limit the block's output to a value between the Lower saturation limit and Upper saturation limit parameters.

- Selecting this check box limits the block output to a value between the Lower saturation limit and Upper saturation limit parameters.
- Clearing this check box does not limit the block output values.

Dependencies

Selecting this parameter enables the Lower saturation limit and Upper saturation limit parameters.

Programmatic Use
Block Parameter: LimitOutput
Type: character vector, string
Values: 'off' | 'on'
Default: 'off'

Upper saturation limit — Upper limit for the integral
inf (default) | scalar | vector | matrix

Specify the upper limit for the integral as a scalar, vector, or matrix. You must specify a value between the Output minimum and Output maximum parameter values.

Dependencies

To enable this parameter, select the Limit output check box.

Programmatic Use
Block Parameter: UpperSaturationLimit
Type: character vector, string
Values: scalar | vector | matrix
Default: 'inf'
Lower saturation limit — Lower limit for the integral

-\infty (default) | scalar | vector | matrix

Specify the lower limit for the integral as a scalar, vector, or matrix. You must specify a value between the Output minimum and Output maximum parameter values.

Dependencies

To enable this parameter, select the Limit output check box.

Programmatic Use

Block Parameter: LowerSaturationLimit
Type: character vector, string
Values: scalar | vector | matrix
Default: '-\infty'

Wrap state — Enable wrapping of states

off (default) | on

Enable wrapping of states between the Wrapped state upper value and Wrapped state lower value parameters. Enabling wrap states eliminates the need for zero-crossing detection, reduces solver resets, improves solver performance and accuracy, and increases simulation time span when modeling rotary and cyclic state trajectories.

If you specify Wrapped state upper value as \infty and Wrapped state lower value as -\infty, wrapping does not occur.

Dependencies

Selecting this parameter enables Wrapped state upper value and Wrapped state lower value parameters.

Programmatic Use

Block Parameter: WrapState
Type: character vector, string
Values: 'off' | 'on'
Default: 'off'

Wrapped state upper value — Specify upper value for wrapped state

\pi (default) | scalar or vector

Upper limit of the block output.
Dependencies

Selecting Wrap state enables this parameter.

Programmatic Use

Block Parameter: WrappedStateUpperValue
Type: scalar or vector
Values: '2*pi'
Default: 'pi'

Wrapped state lower value — Specify lower value for wrap state
-pi (default) | scalar or vector

Lower limit of the block output.

Dependencies

Selecting Wrap state enables this parameter.

Programmatic Use

Block Parameter: WrappedStateLowerValue
Type: scalar or vector
Values: '0'
Default: '-pi'

Show saturation port — Enable saturation output port
off (default) | on

Select this check box to add a saturation output port to the block. When you clear this check box, the block does not have a saturation output port.

Dependencies

Selecting this parameter enables a saturation output port.

Programmatic Use

Block Parameter: ShowSaturationPort
Type: character vector, string
Values: 'off' | 'on'
Default: 'off'

Show state port — Enable state output port
off (default) | on
Select this check box to add a state output port to the block. When you clear this check box, the block does not have a state output port.

**Dependencies**

Selecting this parameter enables a state output port.

**Programmatic Use**

**Block Parameter:** `ShowStatePort`  
**Type:** character vector, string  
**Values:** `'off' | 'on'`  
**Default:** `'off'`

**Absolute tolerance — Absolute tolerance for block states**

`auto` (default) | real scalar or vector

- If you enter `auto` or `-1`, then Simulink uses the absolute tolerance value in the Configuration Parameters dialog box (see “Solver Pane”) to compute block states.
- If you enter a real scalar, then that value overrides the absolute tolerance in the Configuration Parameters dialog box for computing all block states.
- If you enter a real vector, then the dimension of that vector must match the dimension of the continuous states in the block. These values override the absolute tolerance in the Configuration Parameters dialog box.

**Programmatic Use**

**Block Parameter:** `AbsoluteTolerance`  
**Type:** character vector, string, scalar, or vector  
**Values:** `'auto' | '-1' | any positive real scalar or vector`  
**Default:** `'auto'`

**Ignore limit and reset when linearizing — Treat block as unresettable and output unlimited**

`off` (default) | `on`

Cause Simulink linearization commands to treat this block as unresettable and as having no limits on its output, regardless of the settings of the reset and output limitation options of the block.

**Tip**

Use this check box to linearize a model around an operating point that causes the integrator to reset or saturate.
Programmatic Use

**Block Parameter:** IgnoreLimit
**Type:** character vector, string
**Values:** 'off' | 'on'
**Default:** 'off'

Enable zero-crossing detection — Enable zero-crossing detection
**on** (default) | Boolean

Select to enable zero-crossing detection. For more information, see “Zero-Crossing Detection” in the Simulink documentation.

Programmatic Use
**Block Parameter:** ZeroCross
**Type:** character vector, string
**Values:** 'off' | 'on'
**Default:** 'on'

State Name (e.g., 'position') — Assign unique name to each state
' ' (default) | character vector | string

- To assign a name to a single state, enter the name between quotes, for example, 'velocity'.
- To assign names to multiple states, enter a comma-delimited list surrounded by braces, for example, '{a', 'b', 'c}'. Each name must be unique.
- The state names apply only to the selected block.
- The number of states must divide evenly among the number of state names.
- You can specify fewer names than states, but you cannot specify more names than states.

For example, you can specify two names in a system with four states. The first name applies to the first two states and the second name to the last two states.
- To assign state names with a variable in the MATLAB workspace, enter the variable without quotes. A variable can be a character vector, string, cell array, or structure.

Programmatic Use
**Block Parameter:** ContinuousStateAttributes
**Type:** character vector, string
**Values:** ' ' | user-defined
Default: `'

**Block Characteristics**

<table>
<thead>
<tr>
<th>Data Types</th>
<th>double</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>yes</td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>no</td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>yes</td>
</tr>
</tbody>
</table>

**Extended Capabilities**

**C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

- Consider discretizing the model
- Not recommended for production code

**See Also**

Discrete-Time Integrator | Second-Order Integrator

**Introduced before R2006a**
Interpolation Using Prelookup

Use precalculated index and fraction values to accelerate approximation of N-dimensional function

Library: Simulink / Lookup Tables

Description

The Interpolation Using Prelookup block is most efficient when used with the Prelookup block. The Prelookup block calculates the index and interval fraction that specify how its input value \( u \) relates to the breakpoint data set. Feed the resulting index and fraction values into an Interpolation Using Prelookup block to interpolate an \( n \)-dimensional table. These two blocks have distributed algorithms. When combined together, they perform the same operation as the integrated algorithm in the n-D Lookup Table block. However, the Prelookup and Interpolation Using Prelookup blocks offer greater flexibility that can provide more efficient simulation and code generation. For more information, see “Efficiency of Performance”.

Supported Block Operations

To use the Interpolation Using Prelookup block, you specify a set of table data values directly on the dialog box or feed values into the \( T \) input port. Typically, these table values correspond to the breakpoint data sets specified in Prelookup blocks. The Interpolation Using Prelookup block generates output by looking up or estimating table values based on index and interval fraction values fed from Prelookup blocks. Labels for the index and interval fraction appear as \( k \) and \( f \) on the Interpolation Using Prelookup block icon.
When inputs for index and interval fraction... | The Interpolation Using Prelookup block...
--- | ---
Map to values in breakpoint data sets | Outputs the table value at the intersection of the row, column, and higher dimension breakpoints
Do not map to values in breakpoint data sets, but are within range | Interpolates appropriate table values using the **Interpolation method** you select
Do not map to values in breakpoint data sets, and are out of range | Extrapolates the output value using the **Extrapolation method** you select

### How The Block Interpolates a Subset of Table Data

You can use the **Number of sub-table selection dimensions** parameter to specify that interpolation occur only on a subset of the table data. To activate this interpolation mode, set this parameter to a positive integer. This value defines the number of dimensions to select, starting from the highest dimension of table data for the default column-major algorithm. Therefore, the value must be less than or equal to the **Number of table dimensions**.

For row-major algorithms, the interpolation starts from the first or lowest dimension of the table data.

For nonzero values, the subtable selection behavior is optimized for row-major layout when you select the **Math and Data Types > Use algorithms optimized for row-major array layout** configuration parameter.

Suppose that you have 3-D table data in your Interpolation Using Prelookup block. This behavior applies for the column-major algorithm.

<table>
<thead>
<tr>
<th>Number of Selection Dimensions</th>
<th>Action by the Block</th>
<th>Block Appearance</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Interpolates the entire table and does not activate subtable selection</td>
<td>Does not change</td>
</tr>
<tr>
<td>1</td>
<td>Interpolates the first two dimensions and selects the third dimension</td>
<td>Displays an input port with the label s3 that you use to select and interpolate 2-D tables</td>
</tr>
<tr>
<td>Number of Selection Dimensions</td>
<td>Action by the Block</td>
<td>Block Appearance</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>---------------------</td>
<td>------------------</td>
</tr>
<tr>
<td>2</td>
<td>Interpolates the first dimension and selects the second and third dimensions</td>
<td>Displays two input ports with the labels s2 and s3 that you use to select and interpolate 1-D tables</td>
</tr>
</tbody>
</table>

Subtable selection uses zero-based indexing. For an example of interpolating a subset of table data, type `sldemo_bpcheck` at the MATLAB command prompt.

For 2-D or n-D interpolation without subtable selection, the column-major and row-major algorithms may differ in the order of output calculations, causing slight different numerical results.

## Ports

### Input

**k1 — Index, k, for the first dimension of the table**

scalar | vector | matrix

Zero-based index, k, specifying the interval containing the input u for the first dimension of the table.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | fixed point

**f1 — Fraction, f, for the first dimension of the table**

scalar | vector | matrix

Fraction, f, representing the normalized position of the input within the interval, k, for the first dimension of the table.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | fixed point

**kn — Index, k, for the n-th dimension of the table**

scalar | vector | matrix
Zero-based index, \( k \), specifying the interval containing the input \( u \) for the \( n \)-th dimension of the table.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

\( fn \) — Fraction, \( f \), for the \( n \)-th dimension of the table

scalar | vector | matrix

Fraction, \( f \), representing the normalized position of the input within the interval, \( k \), for the \( n \)-th dimension of the table.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

\( kf1 \) — Input containing index \( k \) and fraction \( f \)

bus object

Inputs to the \( kf1 \) port contain index \( k \) and fraction \( f \) specified as a bus object.

Dependencies

To enable this port, select the Require index and fraction as a bus check box.

The number of available \( kf \) input ports depends on the value of the Number of dimensions and Number of sub-table selection dimensions parameters.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point | bus

\( kfn \) — Input containing index \( k \) and fraction \( f \)

bus object

Inputs to the \( kfn \) port contain index \( k \) and fraction \( f \) for the \( n \)-th dimension of the input, specified as a bus object.

Dependencies

To enable this port, select the Require index and fraction as a bus check box.

The number of available \( kf \) input ports depends on the value of the Number of dimensions and Number of sub-table selection dimensions parameters.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point | bus
**sm — Select and interpolate a subset of the table data**

scalar | vector | matrix

The block uses inputs to the \( sm, sm^{-1} \ldots \) port to perform selection and interpolation within the subtables. \( m \) equals the **Number of dimensions - Number of sub-table selection dimensions**.

**Dependencies**

To enable this port, the **Number of sub-table selection dimensions** must be a positive integer less than or equal to the **Number of dimensions**.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus

**T — Table data**

scalar | vector | matrix | n-d array

Table data values provided as input to port \( T \). Typically, these table values correspond to the breakpoint data sets specified in Prelookup blocks. The Interpolation Using Prelookup block generates output by looking up or estimating table values based on index (\( k \)) and interval fraction (\( f \)) values fed from Prelookup blocks.

**Dependencies**

To enable this port, set Source to Input port.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

**Output**

**Port_1 — Approximation of N-dimensional function**

scalar | vector | matrix

Approximation of N-dimensional function, computed by interpolating (or extrapolating) table data using values from the input index, \( k \), and fraction, \( f \).

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point
Parameters

Main

Table data

Number of dimensions — Number of table data dimensions
2 (default) | integer between 1 and 30

Specify the number of dimensions that the table data must have. The Number of dimensions defines the number of independent variables for the table.

<table>
<thead>
<tr>
<th>To specify...</th>
<th>Do this...</th>
</tr>
</thead>
<tbody>
<tr>
<td>1, 2, 3, or 4</td>
<td>Select the value from the drop-down list.</td>
</tr>
<tr>
<td>A higher number of table dimensions</td>
<td>Enter a positive integer directly in the field. The maximum number of table dimensions that this block supports is 30.</td>
</tr>
</tbody>
</table>

Programmatic Use

Block Parameter: NumberOfTableDimensions
Type: character vector
Values: '1' | '2' | '3' | '4' | ... | '30'
Default: '2'

Require index and fraction as bus — Index and fraction inputs can be combined in a bus
off (default) | on

If you select this parameter, subtable selection ports continue to work in nonbus mode.

To enable the Prelookup block to supply input to the Interpolation Using Prelookup block, set:

- Output selection to Index and fraction as bus
- Output to Bus: <object name>, where <object name> must be a valid bus object name accessible to the model

Programmatic Use

Block Parameter: RequireIndexFractionAsBus
**Type**: character vector  
**Values**: 'off' | 'on'  
**Default**: 'off'

**Specification — Choose how to enter table data**
Explicit values (default) | Lookup table object

Specify whether to enter table data directly or use a lookup table object. If you set this parameter to:

- **Explicit values**, the **Source** and **Value** parameters are visible on the dialog box.
- **Lookup table object**, the **Name** parameter is visible on the dialog box.

**Programmatic Use**
**Block Parameter**: TableSpecification  
**Type**: character vector  
**Values**: 'Explicit values' | 'Lookup table object'  
**Default**: 'Explicit values'

**Source — Source of table data**
Dialog (default) | Input port

Specify whether to enter table data in the dialog box or to inherit the data from an input port. If you set **Source** to:

- **Dialog**, enter table data in the text box under **Value**
- **Input port**, verify that an upstream signal supplies table data to the table input port

**Dependencies**
To enable this parameter, set **Specification** to **Explicit values**.

**Programmatic Use**
**Block Parameter**: TableSource  
**Type**: character vector  
**Values**: 'Dialog' | 'Input port'  
**Default**: 'Dialog'

**Value — Specify table data values**
\[ \sqrt{([1:11] \times [1:11])} (default) | \text{multidimensional array of table data} \]

Specify table data as an N-D array, where N is the value of the **Number of dimensions** parameter. You can edit the block diagram without specifying a correctly dimensioned
matrix by entering an empty matrix ([ ]) or an undefined workspace variable in the Value
edit field. For information about how to construct multidimensional arrays in MATLAB,
see “Multidimensional Arrays” (MATLAB).

If you set Source to Input port, verify that an upstream signal supplies table data to
the T input port. The size of table data must match the Number of table dimensions.
For this option, the block inherits table attributes from the T input port.

To edit lookup tables using the Lookup Table Editor, click Edit (see “Edit Lookup Tables”).

Dependencies

To enable this parameter and explicitly specify table values on the dialog box, you must
set Specification to Explicit values and Source to Dialog.

Programmatic Use
Block Parameter: Table
Type: character vector
Values: scalar, vector, matrix, or N-D array
Default: 'sqrt([1:11]' * [1:11])'

Name — Name of a Simulink.LookupTable object
Simulink.LookupTable object

Specify the name of a Simulink.LookupTable object. A lookup table object references
Simulink breakpoint objects. If a Simulink.LookupTable object does not exist, click the
action button and select Create. The corresponding parameters of the new lookup
table object are automatically populated with the block information.

Dependencies

To enable this parameter, set Specification to Lookup table object.

Programmatic Use
Block Parameter: LookupTableObject
Type: character vector
Value: Simulink.LookupTable object
Default: ' '
Algorithm

**Interpolation method — Select Linear point-slope, Flat, Nearest, or Linear Lagrange**
Linear point-slope (default) | Nearest | Flat | Linear Lagrange

Specify the method the block uses to interpolate table data. You can select Linear point-slope, Flat, Nearest, or Linear Lagrange. See “Interpolation Methods” for more information.

**Programmatic Use**

**Block Parameter:** InterpMethod  
**Type:** character vector  
**Values:** 'Flat' | 'Linear point-slope' | 'Nearest' | 'Linear Lagrange'  
**Default:** 'Linear point-slope'

**Extrapolation method — Method of handling input that falls outside the range of the breakpoint data set**
Linear (default) | Clip

Specify the method the block uses to extrapolate values for all inputs that fall outside the range of the breakpoint data set. You can select Clip or Linear. See “Extrapolation Methods” for more information.

If the extrapolation method is Linear, the extrapolation value is calculated based on the selected linear interpolation method. For example, if the interpolation method is Linear Lagrange, the extrapolation method inherits the Linear Lagrange equation to compute the extrapolated value.

**Dependencies**

To enable the **Extrapolation method** parameter, set the **Interpolation method** to Linear.

The Interpolation Using Prelookup block does not support Linear extrapolation when the input or output signals specify integer or fixed-point data types.

**Programmatic Use**

**Block Parameter:** ExtrapMethod  
**Type:** character vector  
**Values:** 'Clip' | 'Linear'  
**Default:** 'Linear'
Valid index input may reach last index — Allow inputs to access the last elements of table data

Specify how block inputs for index (k) and interval fraction (f) access the last elements of n-dimensional table data. Index values are zero based.

This check box is relevant if the input index is equal to or larger than the table's last index for that dimension. Due to rounding, selecting and clearing this check box may result in differing results for the last index between simulation and code generation.

<table>
<thead>
<tr>
<th>Check Box</th>
<th>Block Behavior</th>
</tr>
</thead>
</table>
| on        | Returns the value of the last element in a dimension of its table when:  
            • k indexes the last table element in the corresponding dimension  
            • f is 0          |
| off       | Returns the value of the last element in a dimension of its table when:  
            • k indexes the next-to-last table element in the corresponding dimension  
            • f is 1          |

Dependencies

This check box is visible only when:

- Interpolation method is Linear point-slope or Linear Lagrange
- Extrapolation method is Clip

Tip When you select Valid index input may reach last index for an Interpolation Using Prelookup block, you must also select Use last breakpoint for input at or above upper limit for all Prelookup blocks that feed it. This action allows the blocks to use the same indexing convention when accessing the last elements of their breakpoint and table data sets.
**Programmatic Use**

**Block Parameter:** `ValidIndexMayReachLast`  
**Type:** character vector  
**Values:** 'off' | 'on'  
**Default:** 'off'

**Diagnostic for out-of-range input — Block action when input is out of range**

None (default) | Warning | Error  

Specify whether to produce a warning or error when the input is out of range. Options include:

- None — Produce no response.
- Warning — Display a warning and continue the simulation.
- Error — Terminate the simulation and display an error.

**Programmatic Use**

**Block Parameter:** `DiagnosticForOutOfRangeInput`  
**Type:** character vector  
**Values:** 'None' | 'Warning' | 'Error'  
**Default:** 'None'

**Number of sub-table selection dimensions — Number of dimensions of the output computation subtable**

0 (default) | positive integer, less than or equal to the number of table dimensions

Specify the number of dimensions of the subtable that the block uses to compute the output. Follow these rules:

- To enable subtable selection, enter a positive integer.
  
  This integer must be less than or equal to the **Number of table dimensions**.

- To disable subtable selection, enter 0 to interpolate the entire table.

For nonzero values, the subtable selection behavior is optimized for row-major layout when you select the **Math and Data Types > Use algorithms optimized for row-major array layout** configuration parameter.

For more information, see “How The Block Interpolates a Subset of Table Data” on page 1-1054.
Programmatic Use

**Block Parameter:** NumSelectionDims

**Type:** character vector

**Values:** '0' | '1' | '2' | '3' | '4' | ... | Number of table dimensions

**Default:** '0'

Code generation

**Remove protection against out-of-range index in generated code** —
Remove code that checks for out-of-range index inputs

off (default) | on

<table>
<thead>
<tr>
<th>Check Box</th>
<th>Result</th>
<th>When to Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>on</td>
<td>Generated code does not include conditional statements to check for out-of-range index inputs. When the input k or f is out of range, it may cause undefined behavior for generated code and simulations using accelerator mode.</td>
<td>For code efficiency</td>
</tr>
<tr>
<td>off</td>
<td>Generated code includes conditional statements to check for out-of-range index inputs.</td>
<td>For safety-critical applications</td>
</tr>
</tbody>
</table>

If your input is not out of range, you can select the **Remove protection against out-of-range index in generated code** check box for code efficiency. By default, this check box is cleared. For safety-critical applications, do not select this check box. If you want to select the **Remove protection against out-of-range index in generated code** check box, first check that your model inputs are in range. For example:

1. Clear the **Remove protection against out-of-range index in generated code** check box.
2. Set the **Diagnostic for out-of-range input** parameter to **Error**.
3. Simulate the model in normal mode.
If there are out-of-range errors, fix them to be in range and run the simulation again.

When the simulation no longer generates out-of-range input errors, select the **Remove protection against out-of-range index in generated code** check box.

**Note** When you select the **Remove protection against out-of-range index in generated code** check box and the input \( k \) or \( f \) is out of range, the behavior is undefined for generated code and simulations using accelerator mode.

Depending on your application, you can run the following Model Advisor checks to verify the usage of this check box:

- **By Product > Embedded Coder > Identify lookup table blocks that generate expensive out-of-range checking code**
- **By Product > Simulink Check > Modeling Standards > DO-178C/DO-331 Checks > Check usage of lookup table blocks**

For more information about the Model Advisor, see “Run Model Advisor Checks”.

This check box has no effect on the generated code when one of the following is true:

- The Prelookup block feeds index values to the Interpolation Using Prelookup block. Because index values from the Prelookup block are always valid, no check code is necessary.
- The data type of the input \( k \) restricts the data to valid index values. For example, unsigned integer data types guarantee nonnegative index values. Therefore, unsigned input values of \( k \) do not require check code for negative values.

**Programmatic Use**

**Block Parameter:** RemoveProtectionIndex  
**Type:** character vector  
**Values:** 'off' | 'on'  
**Default:** 'off'

**Sample time — Specify sample time as a value other than -1**

-1 (default) | scalar | vector

Specify the sample time as a value other than -1. For more information, see “Specify Sample Time”.

1-1065
Dependencies

This parameter is not visible unless it is explicitly set to a value other than -1. To learn more, see “Blocks for Which Sample Time Is Not Recommended”.

Programmatic Use

Block Parameter: SampleTime
Type: character vector
Values: scalar or vector
Default: '-1'

Data Types

Table data — Data type of table values

Specify the table data type. You can set it to:

- A rule that inherits a data type, for example, Inherit: Same as output
- The name of a built-in data type, for example, single
- The name of a data type object, for example, a Simulink.NumericType object
- An expression that evaluates to a data type, for example, fixdt(1,16,0)

Tip Specify a table data type different from the output data type for these cases:

- Lower memory requirement for storing table data that uses a smaller type than the output signal
- Sharing of prescaled table data between two Interpolation Using Prelookup blocks with different output data types
- Sharing of custom storage table data in Simulink Coder generated code for blocks with different output data types
Programmatic Use

Block Parameter: TableDataTypeStr
Type: character vector
Values: 'Inherit: Inherit from table data' | 'Inherit: Same as output' | 'double' | 'single' | 'int8' | 'uint8' | 'int16' | 'uint16' | 'int32' | 'uint32' | 'int64' | 'uint64' | 'fixdt(1,16)' | 'fixdt(1,16,0)' | 'fixdt(1,16,2^0,0)' | '<data type expression>'
Default: 'Inherit: Same as input'

Table minimum — Minimum value of table data
[] (default) | scalar

Specify the minimum value for table data as a finite, real, double, scalar. The default value is [] (unspecified).

Programmatic Use

Block Parameter: TableMin
Type: character vector
Values: scalar
Default: '[]'

Table maximum — Maximum value of table data
[] (default) | scalar

Specify the maximum value for table data as a finite, real, double, scalar. The default value is [] (unspecified).

Programmatic Use

Block Parameter: TableMax
Type: character vector
Values: scalar
Default: '[]'

Intermediate results — Data type of intermediate results

Inherit: Inherit via internal rule (default) | Inherit: Same as output | double | single | int8 | uint8 | int16 | uint16 | int32 | uint32 | int64 | uint64 | fixdt(1,16) | fixdt(1,16,0) | fixdt(1,16,2^0,0) | <data type expression>

Specify the intermediate results data type. You can set it to:

- A rule that inherits a data type, for example, Inherit: Same as output
- The name of a built-in data type, for example, single
• The name of a data type object, for example, a `Simulink.NumericType` object
• An expression that evaluates to a data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button to display the **Data Type Assistant**, which helps you set the data type attributes. For more information, see “Specify Data Types Using Data Type Assistant”.

**Tip** Use this parameter to specify higher precision for internal computations than for table data or output data.

**Programmatic Use**

**Block Parameter:** `IntermediateResultsDataTypeStr`

**Type:** character vector

**Values:**
- 'Inherit: Inherit via internal rule'
- 'Inherit: Same as output'
- 'double'
- 'single'
- 'int8'
- 'uint8'
- 'int16'
- 'uint16'
- 'int32'
- 'uint32'
- 'int64'
- 'uint64'
- 'fixdt(1,16)'
- 'fixdt(1,16,0)'
- 'fixdt(1,16,2^0,0)'
- '<data type expression>'

**Default:** 'Inherit: Same as input'

**Output — Output data type**

Inherit: Inherit from 'Table data' (default) | Inherit: Inherit via back propagation | double | single | int8 | uint8 | int16 | uint16 | int32 | uint32 | int64 | uint64 | fixdt(1,16) | fixdt(1,16,0) | fixdt(1,16,2^0,0) | <data type expression>

Specify the output data type. You can set it to:

• A rule that inherits a data type, for example, `Inherit: Inherit via back propagation`
• The name of a built-in data type, for example, `single`
• The name of a data type object, for example, `a Simulink.NumericType` object
• An expression that evaluates to a data type, for example, `fixdt(1,16,0)`

Click the **Show data type assistant** button to display the **Data Type Assistant**, which helps you set the output data type.

See “Control Signal Data Types” in the *Simulink User's Guide* for more information.
Programmatic Use
Block Parameter: OutDataTypeStr
Type: character vector
Values: 'Inherit: Inherit via back propagation' | 'Inherit: Inherit from table data' | 'double' | 'single' | 'int8' | 'uint8' | 'int16' | 'uint16' | 'int32' | 'uint32' | 'int64' | 'uint64' | 'fixdt(1,16,0)' | 'fixdt(1,16,2^0,0)' | 'fixdt(1,16,2^0,0)' | '<data type expression>'
Default: 'Inherit: Inherit from table data'

Output minimum — Minimum value the block can output
[] (default) | scalar

Specify the minimum value that the block should output as a finite, real-valued scalar. The default value is [] (unspecified). Simulink software uses this value to perform:

- Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters”).
- Simulation range checking (see “Signal Ranges”).
- Automatic scaling of fixed-point data types.
- Optimization of the code that you generate from the model. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. For more information, see “Optimize using the specified minimum and maximum values” (Simulink Coder).

Programmatic Use
Block Parameter: OutMin
Type: character vector
Values: scalar
Default: ['']

Output maximum — Maximum value the block can output
[] (default) | scalar

Specify the maximum value that the block should output as a finite, real-valued scalar. The default value is [] (unspecified). Simulink software uses this value to perform:

- Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters”).
- Simulation range checking (see “Signal Ranges”).
Automatic scaling of fixed-point data types.

Optimization of the code that you generate from the model. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. For more information, see “Optimize using the specified minimum and maximum values” (Simulink Coder).

Programmatic Use

Block Parameter: OutMax
Type: character vector
Values: scalar
Default: '[]'

Internal rule priority — Internal rule for intermediate calculations
Speed (default) | Precision

Specify the internal rule for intermediate calculations. Select Speed for faster calculations. If you do, a loss of accuracy might occur, usually up to 2 bits.

Programmatic Use

Block Parameter: InternalRulePriority
Type: character vector
Values: 'Speed' | 'Precision'
Default: 'Speed'

Lock data type settings against changes by the fixed-point tools — Prevent fixed-point tools from overriding data types
off (default) | on

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on this block. For more information, see “Lock the Output Data Type Setting” (Fixed-Point Designer).

Programmatic Use

Block Parameter: LockScale
Type: character vector
Values: 'off' | 'on'
Default: 'off'

Integer rounding mode — Rounding mode for fixed-point operations
Floor (default) | Convergent | Ceiling | Nearest | Round | Simplest | Zero

Specify the rounding mode for fixed-point operations. For more information, see “Rounding” (Fixed-Point Designer).
Block parameters always round to the nearest representable value. To control the rounding of a block parameter, enter an expression using a MATLAB rounding function in the mask field.

**Programmatic Use**

**Block Parameter:** RndMeth  
**Type:** character vector  
**Values:** 'Ceiling' | 'Convergent' | 'Floor' | 'Nearest' | 'Round' | 'Simplest' | 'Zero'  
**Default:** 'Floor'

**Saturate on integer overflow — Method of overflow action**

off (default) | on

Specify whether overflows saturate or wrap.

<table>
<thead>
<tr>
<th>Action</th>
<th>Rationale</th>
<th>Impact on Overflows</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Select this check box (on).</td>
<td>Your model has possible overflow, and you want explicit saturation protection in the generated code.</td>
<td>Overflows saturate to either the minimum or maximum value that the data type can represent.</td>
<td>The maximum value that the int8 (signed, 8-bit integer) data type can represent is 127. Any block operation result greater than this maximum value causes overflow of the 8-bit integer. With the check box selected, the block output saturates at 127. Similarly, the block output saturates at a minimum output value of -128.</td>
</tr>
</tbody>
</table>
### Action | Rationale | Impact on Overflows | Example
--- | --- | --- | ---
Do not select this check box (off). | You want to optimize efficiency of your generated code. You want to avoid overspecifying how a block handles out-of-range signals. For more information, see “Check for Signal Range Errors”. | Overflows wrap to the appropriate value that is representable by the data type. | The maximum value that the int8 (signed, 8-bit integer) data type can represent is 127. Any block operation result greater than this maximum value causes overflow of the 8-bit integer. With the check box cleared, the software interprets the overflow-causing value as int8, which can produce an unintended result. For example, a block result of 130 (binary 1000 0010) expressed as int8, is -126. When you select this check box, saturation applies to every internal operation on the block, not just the output, or result. Usually, the code generation process can detect when overflow is not possible. In this case, the code generator does not produce saturation code.

**Programmatic Use**

**Block Parameter:** SaturateOnIntegerOverflow

**Type:** character vector

**Values:** 'off' | 'on'

**Default:** 'off'

### Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>double</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

PLC Code Generation
Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

See Also
Prelookup

Topics
“About Lookup Table Blocks”
“Anatomy of a Lookup Table”
“Enter Breakpoints and Table Data”
“Guidelines for Choosing a Lookup Table”
“Column-Major Layout to Row-Major Layout Conversion of Models with Lookup Table Blocks” (Simulink Coder)
“Interpolation with Subtable Selection Algorithm for Row-Major Array Layout” (Simulink Coder)

Introduced in R2006b
Interpreted MATLAB Function

Apply MATLAB function or expression to input
Library: Simulink / User-Defined Functions

Description

The Interpreted MATLAB Function block applies the specified MATLAB function or expression to the input. The output of the function must match the output dimensions of the block.

Some valid expressions for this block are:

\[
\begin{align*}
\sin u(1), \ u(2) \\
\text{atan2}(u(1), u(2)) \\
u(1)^u(2)
\end{align*}
\]

**Note** This block is slow because it calls the MATLAB parser during each integration step. Consider using built-in blocks (such as the Math Function block) instead. Alternatively, you can write the function as a MATLAB S-function or MEX-file S-function, then access it using the S-Function block.

Ports

**Input**

*In — Input of a Interpreted MATLAB function*

**scalar | vector | matrix**

The Interpreted MATLAB Function block accepts one real or complex input of type `double` and generates real or complex output of type `double`, depending on the setting of the **Output signal type** parameter.
Data Types: double

Output

Out — Output of a Interpreted MATLAB function

scalar | vector | matrix

The Interpreted MATLAB Function block accepts one real or complex input of type double and generates real or complex output of type double, depending on the setting of the Output signal type parameter.

Data Types: double

Parameters

MATLAB Function — Specify the function or expression

function (default)

Specify the function or expression. If you specify a function only, it is not necessary to include the input argument in parentheses.

Output dimensions — Specify the dimensions of the block output signal

scalar | matrix | vector | inherited

Specify the dimensions of the block output signal, for example, 2 for a two-element vector. The output dimensions must match the dimensions of the value returned by the function or expression in the MATLAB function field.

Specify -1 to inherit the dimensions from the output of the specified function or expression. To determine the output dimensions, Simulink runs the function or expression once before simulation starts.

Note If you specify -1 for this parameter and your function has persistent variables, then the variables might update before the simulation starts. If you need to use persistent variables, consider setting this parameter to a value other than -1.

Output signal type — Specify the block output signal type

auto (default) | real | complex
Specify the output signal type of the block as **real**, **complex**, or **auto**. A value of **auto** sets the output type to be the same as the type of the input signal.

**Collapse 2-D results to 1-D — Output a 2-D array as a 1-D array**

- **off** (default) | **on**

Select this check box to output a 2-D array as a 1-D array containing the 2-D array's elements in column-major order.

**Sample time — Specify sample time in the block**

- **scalar**

**Note** This parameter is not visible in the block dialog box unless it is explicitly set to a value other than -1. To learn more, see “Blocks for Which Sample Time Is Not Recommended”.

### Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>double</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>yes</td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>no</td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
</tr>
</tbody>
</table>

### See Also

MATLAB Function | MATLAB System

**Introduced in R2011a**
Interval Test

Determine if signal is in specified interval

Library: Simulink / Logic and Bit Operations

Description

The Interval Test block outputs true (1) if the input is between the values specified by the Lower limit and Upper limit parameters. The block outputs false (0) if the input is outside those values. The output of the block when the input is equal to the Lower limit or the Upper limit is determined by whether you select the Interval closed on left and Interval closed on right check boxes.

Ports

Input

Port_1 — Input signal
scalar | vector | matrix | N-D array

Input signal, specified as a scalar, vector, matrix, or N-D array.

Limitations

When the input signal is an enumerated type, the Upper limit and Lower limit values must be of the same enumerated type.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated

Output

Port_1 — Output signal
scalar | vector | matrix | N-D array
Output signal indicating whether the input values fall within the specified interval. You can specify the **Output data type** as boolean or uint8.

Data Types: uint8 | Boolean

## Parameters

**Interval closed on right — Include upper limit value**

*on* (default) | *off*

When you select this check box, the **Upper limit** is included in the interval for which the block outputs true (1).

**Programmatic Use**

**Block Parameter:** IntervalClosedRight  
**Type:** character vector  
**Values:** 'on' | 'off'  
**Default:** 'on'

**Upper limit — Upper limit of interval**

0.5 (default) | scalar | vector | matrix | N-D array

The upper limit of the interval for which the block outputs true (1).

**Programmatic Use**

**Block Parameter:** uplimit  
**Type:** character vector  
**Values:** scalar | vector | matrix | N-D array  
**Default:** '0.5'

**Interval closed on left — Include lower limit value**

*on* (default) | *off*

When you select this check box, the **Lower limit** is included in the interval for which the block outputs true (1).

**Programmatic Use**

**Block Parameter:** IntervalClosedLeft  
**Type:** character vector  
**Values:** 'on' | 'off'  
**Default:** 'on'
**Lower limit — Lower limit of interval**
-0.5 (default) | scalar | vector | matrix | N-D array

The lower limit of the interval for which the block outputs true (1).

**Programmatic Use**
**Block Parameter:** lowlimit  
**Type:** character vector  
**Values:** scalar | vector | matrix | N-D array  
**Default:** '-0.5'

**Output data type — Output data type**
boolean (default) | uint8

Specify the output data type as boolean or uint8.

**Programmatic Use**
**Block Parameter:** OutDataTypeStr  
**Type:** character vector  
**Values:** 'boolean' | 'uint8'  
**Default:** 'boolean'

**Block Characteristics**

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>double</th>
<th>enumerated</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Direct Feedthrough</th>
<th>yes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multidimensional Signals</td>
<td>yes</td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
</tr>
</tbody>
</table>
Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

PLC Code Generation
Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

See Also
Interval Test Dynamic

Introduced before R2006a
Interval Test Dynamic

Determine if signal is in specified interval
Library: Simulink / Logic and Bit Operations

Description

The Interval Test Dynamic block outputs true (1) if the input is between the values of the external signals up and lo. The block outputs false (0) if the input is outside those values. To control how the block handles input values that are equal to the signal lo or the signal up, use the Interval closed on left and Interval closed on right check boxes.

Ports

Input

up — Upper limit of interval
scalar | vector | matrix | N-D array

Upper limit of interval, specified as a scalar, vector, matrix, or N-D array.

Limitations

When the input signal is an enumerated type, the up and lo signals must be of the same enumerated type.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated

u — Input signal
scalar | vector | matrix | N-D array

Input signal, specified as a scalar, vector, matrix, or N-D array.
Limitations

When the input signal is an enumerated type, the **up** and **lo** signals must be of the same enumerated type.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `Boolean` | `fixed point` | `enumerated`

**lo — Lower limit of interval**

scalar | vector | matrix | N-D array

Lower limit of interval, specified as a scalar, vector, matrix, or N-D array.

Limitations

When the input signal is an enumerated type, the **up** and **lo** signals must be of the same enumerated type.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `Boolean` | `fixed point` | `enumerated`

**Output**

**y — Output signal**

scalar | vector | matrix | N-D array

Output signal indicating whether the input values fall within the specified interval. You can specify the **Output data type** as `boolean` or `uint8`.

Data Types: `uint8` | `Boolean`

**Parameters**

**Interval closed on right — Include upper limit value**

on (default) | off

When you select this check box, the value of the signal connected to the **up** input port is included in the interval for which the block outputs true (1).

**Programmatic Use**

**Block Parameter:** `IntervalClosedRight`

**Type:** character vector
Values: 'on' | 'off'
Default: 'on'

Interval closed on left — Include lower limit value
on (default) | off

When you select this check box, the value of the signal connected to the lo input port is included in the interval for which the block outputs true (1).

Programmatic Use
Block Parameter: IntervalClosedLeft
Type: character vector
Values: 'on' | 'off'
Default: 'on'

Output data type — Output data type
boolean (default) | uint8

Specify the output data type as boolean or uint8.

Programmatic Use
Block Parameter: OutDataTypeStr
Type: character vector
Values: 'boolean' | 'uint8'
Default: 'boolean'

Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>double</th>
<th>enumerated</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

PLC Code Generation
Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

See Also
Interval Test

Introduced before R2006a
**Description**

The Knob block tunes the value of the connected block parameter to during simulation. For example, you can connect the Knob block to a Gain block in your model and adjust its value during simulation. You can modify the range of the Knob block's scale to fit your data. Use the Knob block with other Dashboard blocks to create an interactive dashboard to control your model.

**Connecting Dashboard Blocks**

Dashboard blocks do not use ports to make connections. To connect Dashboard blocks to variables and block parameters in your model, use connect mode. Connect mode facilitates the process of connecting Dashboard blocks in your model, especially when you want to connect multiple blocks at once. If you only want to connect a single Dashboard block, you can also use the **Connection** table in the block dialog box to make the connection.

**Tip** To facilitate understanding and debugging your model, you can modify Dashboard block connections in your model during normal and accelerator mode simulations.

**Note** Dashboard blocks cannot connect to variables until you update your model diagram. To connect Dashboard blocks to variables or modify variable values between opening your model and running a simulation, update your model diagram using **Ctrl+D**.
To enter connect mode, click the **Connect** button that appears above your unconnected Dashboard block when you pause on it.

In connect mode, when you select one or more signals or blocks, a list of parameters and signals available for connection appears. Select a signal or parameter from the list to connect the selected Dashboard block. To connect another Dashboard block, pause on the block and click the **Connect** button above it. Then, make a selection of signals and blocks in your model, and choose a signal or parameter to connect.
When you finish connecting the Dashboard blocks in your model, click the **Exit** button in the upper-right of the canvas to exit connect mode.

**Tip** You can hide the message shown on unconnected blocks using the `set_param` function with the `ShowInitialText` block parameter.

**Parameter Logging**

Tunable parameters connected to Dashboard blocks are logged to the Simulation Data Inspector, where you can view parameter values along with logged signal data. You can access logged parameter data in the MATLAB workspace by exporting the parameter data from the Simulation Data Inspector UI or by using the `Simulink.sdi.exportRun` function. For more information about exporting data with the Simulation Data Inspector UI, see “Export Data from the Simulation Data Inspector”. The parameter data is stored in a `Simulink.SimulationData.Parameter` object, accessible as an element in the exported `Simulink.SimulationData.Dataset`. 
Limitations

- Dashboard blocks can only connect to real scalar signals.
- You cannot use the **Connection** table to connect a Dashboard block to a block that is commented out. When you connect a Dashboard block to a commented block using connect mode, the Dashboard block does not display the connected value until the you uncomment the block.
- Dashboard blocks cannot connect to signals inside referenced models.
- When you simulate a model hierarchy, Dashboard blocks inside referenced models do not update.
- Dashboard blocks do not support rapid accelerator simulation.
- Some signals are not available during accelerator mode simulations. To view such signals on a Dashboard block, mark the signal for logging.
- Parameters specified by indexing a variable array do not appear in the **Connection** table. For example, a block parameter defined as `engine(1)` using the variable `engine` does not appear in the table.

To access the parameter in the **Connection** table, assign the indexed value to a scalar variable, such as `engine_1`. Then, use the scalar variable to define the block parameter.

- When you connect a Dashboard block to a variable or parameter during simulation, the data for that variable or parameter does not get logged to the Simulation Data Inspector. To log variable and parameter data to the Simulation Data Inspector, connect the Dashboard block to the variable or parameter prior to simulation.
- When you simulate a model in external mode with the **Default parameter behavior** set to **Inlined**, Dashboard blocks can appear to change parameter and variable values. However, the change does not propagate to the simulation. For example, the Gain block displays changes made to its **Gain** parameter using the Dashboard blocks, but the **Gain** value used in the simulation does not change.

Parameters

**Connection** — Select a variable or block parameter to connect
variable and parameter connection options

Select the variable or block parameter to control using the **Connection** table. Populate the **Connection** table by selecting one or more blocks in your model. Select the radio
button next to the variable or parameter you want to control, and click **Apply**. To facilitate understanding and debugging your model, you can connect Dashboard blocks to variables and parameters in your model during simulation.

**Note** To see workspace variables in the connection table, update the model diagram using **Ctrl+D**.

**Programmatic Use**

To programmatically connect a Dashboard block to a tunable parameter or a variable, use a `Simulink.HMI.ParamSourceInfo` object. The `Simulink.HMI.ParamSourceInfo` object contains four properties. Some of the properties apply to connecting Dashboard blocks to parameters, and some apply to connecting Dashboard blocks to variables. Not all fields have a value for a connection because a given Dashboard block connects to either a parameter or a variable.

**Block Parameter:** Binding  
**Type:** `Simulink.HMI.ParamSourceInfo`  
**Default:** `[]`

**Scale Type — Type of scale**

*Linear* (default) | *Log*

Type of scale displayed on the control. **Linear** specifies a linear scale, and **Log** specifies a logarithmic scale.

**Programmatic Use**

**Block Parameter:** ScaleType  
**Type:** string or character vector  
**Values:** 'Linear' | 'Log'  
**Default:** 'Linear'

**Minimum — Minimum tick mark value**

0 (default) | scalar

Finite, real, double, scalar value specifying the minimum tick mark value for the arc. The minimum must be less than the value entered for the maximum.

**Programmatic Use**

To programatically set the Minimum parameter, use a 1-by-3 vector containing values for the Minimum, Tick Interval, and Maximum parameters, in that order. To use the
auto value for the **Tick Interval**, leave the **Tick Interval** position in the vector empty or specify -1.

**Block Parameter:** Limits
**Type:** 1x3 vector
**Default:** [0 -1 100]

**Maximum — Maximum tick mark value**

100 (default) | scalar

Finite, real, double, scalar value specifying the maximum tick mark value for the arc. The maximum must be greater than the value entered for the minimum.

**Programmatic Use**

To programmatically set the **Maximum** parameter, use a 1-by-3 vector containing values for the **Minimum**, **Tick Interval**, and **Maximum** parameters, in that order. To use the auto value for the **Tick Interval**, leave the **Tick Interval** position in the vector empty or specify -1.

**Block Parameter:** Limits
**Type:** 1x3 vector
**Default:** [0 -1 100]

**Tick Interval — Interval between major tick marks**

auto (default) | scalar

Finite, real, positive, integer, scalar value specifying the interval of major tick marks on the arc. When set to auto, the block automatically adjusts the tick interval based on the minimum and maximum values.

**Programmatic Use**

To programmatically set the **Tick Interval** parameter, use a 1-by-3 vector containing values for the **Minimum**, **Tick Interval**, and **Maximum** parameters, in that order. To use the auto value for the **Tick Interval**, leave the **Tick Interval** position in the vector empty or specify -1.

**Block Parameter:** Limits
**Type:** 1x3 vector
**Default:** [0 -1 100]

**Label — Block label position**

Top (default) | Bottom | Hide
Position of the block label. When the block is connected to an element in the model, the label is the name of the connected element.

**Programmatic Use**
**Block Parameter:** LabelPosition  
**Type:** character vector  
**Values:** 'Top' | 'Bottom' | 'Hide'  
**Default:** 'Top'

### Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>double</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>no</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Extended Capabilities

**C/C++ Code Generation**
Generate C and C++ code using Simulink® Coder™.

Dashboard blocks are ignored for code generation.

**See Also**
Rotary Switch | Slider
Topics
“Tune and Visualize Your Model with Dashboard Blocks”
“Decide How to Visualize Simulation Data”

Introduced in R2015a
Lamp

Display color reflecting input value

Library: Simulink / Dashboard

Description

The Lamp block displays a color indicating the value of the input signal. You can use the Lamp block with other Dashboard blocks to build an interactive dashboard of controls and indicators for your model. You can specify pairs of input values and colors to provide the information you want during simulation.

Connecting Dashboard Blocks

Dashboard blocks do not use ports to connect to signals. To connect Dashboard blocks to signals in your model, use connect mode. Connect mode facilitates the process of connecting Dashboard blocks in your model, especially when you want to connect multiple blocks at once. If you only want to connect a single Dashboard block, you can also use the Connection table in the block dialog box to make the connection.

Tip
To facilitate understanding and debugging your model, you can modify Dashboard block connections in your model during normal and accelerator mode simulations.

To enter connect mode, click the Connect button that appears above your unconnected Dashboard block when you pause on it.
In connect mode, when you select one or more signals or blocks, a list of parameters and signals available for connection appears. Select a signal or parameter from the list to connect the selected Dashboard block. To connect another Dashboard block, pause on the block and click the **Connect** button above it. Then, make a selection of signals and blocks in your model, and choose a signal or parameter to connect.
When you finish connecting the Dashboard blocks in your model, click the **Exit** button in the upper-right of the canvas to exit connect mode.

**Tip** You can hide the message shown on unconnected blocks using the `set_param` function and the `ShowInitialText` block parameter.

## Limitations

- Dashboard blocks can only connect to real scalar signals.
- You cannot use the **Connection** table to connect a Dashboard block to a block that is commented out. When you connect a Dashboard block to a commented block using connect mode, the Dashboard block does not display the connected value until the you uncomment the block.
- Dashboard blocks cannot connect to signals inside referenced models.
- When you simulate a model hierarchy, Dashboard blocks inside referenced models do not update.
- Dashboard blocks do not support rapid accelerator simulation.
Some signals are not available during accelerator mode simulations. To view such signals on a Dashboard block, mark the signal for logging.

**Parameters**

**Connection — Select a signal to connect and display**

signal connection options

Select the signal to connect using the **Connection** table. Populate the **Connection** table by selecting signals of interest in your model. Select the radio button next to the signal you want to display. Click **Apply** to connect the signal. To facilitate understanding and debugging your model, you can connect Dashboard blocks to signals in your model during simulation.

**Programmatic Use**

**Block Parameter:** Binding  
**Type:** Simulink.HMI.SignalSpecification  
**Default:** []

**States — Match values to colors to display**

scalar and RGB vector

States match a display color to the value of the connected variable or parameter. Each state consists of a **State** and a **Color**.

- **State** — Connected variable or parameter value that causes the Lamp block to display the specified color indication.
- **Color** — Lamp color when the connected variable or parameter value matches the corresponding **State** value. You can select from a palette of standard colors or specify a custom color with RGB values.

The [undefined] state specifies the color of the Lamp block when the input value does not match any of the values specified in the **States** table. Click the + button to add another state.

**Programmatic Use**

To programatically specify the State and Color parameter values for a Lamp block, use an array of structures containing the fields:
• **Value** — Scalar double value corresponding to the state.
• **Color** — 1-by-3 [r g b] vector with values between 0 and 1 specifying the color for the Lamp to display.

```matlab
lampState1.Value = 1;
lampstate1.Color = [0 0 1];
lampState2.Value = 2;
lampState2.Color = [1 0 0];
lampStates = [lampState1 lampState2];
```

**Block Parameter:** StateColors  
**Type:** structure array  
**Default:** structure

The ColorDefault parameter specifies the color for the Lamp block when the value of the connected signal does not match any of the specified state values. Specify the ColorDefault parameter as a 1-by-3 [r g b] vector with values between 0 and 1.

**Block Parameter:** ColorDefault  
**Type:** [r g b] vector  
**Default:** [0.7529 0.7529 0.7529]

**Label — Block label position**  
**Top** (default) | **Bottom** | **Hide**

Position of the block label. When the block is connected to an element in the model, the label is the name of the connected element.

**Programmatic Use**  
**Block Parameter:** LabelPosition  
**Type:** character vector  
**Values:** 'Top' | 'Bottom' | 'Hide'  
**Default:** 'Top'

**Block Characteristics**

<table>
<thead>
<tr>
<th><strong>Data Types</strong></th>
<th>Boolean</th>
<th>double</th>
<th>enumerated</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Direct Feedthrough</strong></td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Multidimensional Signals | no |
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
</tr>
</tbody>
</table>

**Extended Capabilities**

**C/C++ Code Generation**
Generate C and C++ code using Simulink® Coder™.

Dashboard blocks are ignored for code generation.

**See Also**
MultiStateImage

**Topics**
“Tune and Visualize Your Model with Dashboard Blocks”
“Decide How to Visualize Simulation Data”

**Introduced in R2015a**
Level-2 MATLAB S-Function

Use Level-2 MATLAB S-function in model
Library: Simulink / User-Defined Functions

Description

This block allows you to use a Level-2 MATLAB S-function (see “Write Level-2 MATLAB S-Functions”) in a model. To do this, create an instance of this block in the model. Then enter the name of the Level-2 MATLAB S-function in the S-function name field of the block's parameter dialog box.

Note Use the S-Function block to include a Level-1 MATLAB S-function in a block.

If the Level-2 MATLAB S-function defines any additional parameters, you can enter them in the Parameters field of the block's parameter dialog box. Enter the parameters as MATLAB expressions that evaluate to their values in the order defined by the MATLAB S-function. Use commas to separate each expression.

If a model includes a Level-2 MATLAB S-Function block, and an error occurs in the S-function, the Level-2 MATLAB S-Function block displays MATLAB stack trace information for the error in a dialog box. Click OK to close the dialog box.

Parameters

S-Function Name — Specify S-Function name
matlabfile (default) | S-Function name

Specify the name of a MATLAB function that defines the behavior of this block. The MATLAB function must follow the Level-2 standard for writing MATLAB S-functions (see “Write Level-2 MATLAB S-Functions” for details).
Programmatic Use
Block Parameter: FunctionName
Type: character vector
Values: 'matlabfile' | S-Function name
Default: 'matlabfile'

Parameters — Specify values of parameters for this block
no default (default)

Specify values of parameters for this block.

Programmatic Use
Block Parameter: Parameters
Type: character vector
Values: values of block parameters
Default: ' ' 

Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean^a</th>
<th>double^a</th>
<th>fixed point^a</th>
<th>integer^a</th>
<th>single^a</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

^a. Level-1 MATLAB S-functions support only the double data type. Level-2 MATLAB S-functions support all data types that Simulink supports.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.
• If a corresponding TLC file is available, the Level-2 MATLAB S-Function block uses the TLC file to generate code, otherwise code generation throws an error.
• Real-time code generation does not support S-functions that call MATLAB.
• Actual data type or capability support depends on block implementation.

Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

Level-1 MATLAB S-functions support only the double data type. Level-2 MATLAB S-functions support all data types that Simulink supports.

See Also

Blocks
MATLAB Function | MATLAB System | S-Function | S-Function Builder | Simulink Function | Subsystem, Atomic Subsystem, Nonvirtual Subsystem, CodeReuse Subsystem

Topics
“What Is an S-Function?”
“Write Level-2 MATLAB S-Functions”
“Design and Create a Custom Block”

Introduced in R2010b
Linear Gauge

Display input value on linear scale

**Library:** Simulink / Dashboard

---

**Description**

The Linear Gauge block displays the connected signal on a straight linear scale during simulation. Use the Linear Gauge block with other Dashboard blocks to build an interactive dashboard of controls and indicators for your model. The Linear Gauge block provides an indication of the instantaneous value of the connected signal throughout simulation. You can modify the range of the Linear Gauge block to fit your data. You can also customize the appearance of the Linear Gauge block to provide more information about your signal. For example, you can color-code in-specification and out-of-specification ranges.

---

**Connecting Dashboard Blocks**

Dashboard blocks do not use ports to connect to signals. To connect Dashboard blocks to signals in your model, use connect mode. Connect mode facilitates the process of connecting Dashboard blocks in your model, especially when you want to connect multiple blocks at once. If you only want to connect a single Dashboard block, you can also use the **Connection** table in the block dialog box to make the connection.

---

**Tip**

To facilitate understanding and debugging your model, you can modify Dashboard block connections in your model during normal and accelerator mode simulations.

---

To enter connect mode, click the **Connect** button that appears above your unconnected Dashboard block when you pause on it.
In connect mode, when you select one or more signals or blocks, a list of parameters and signals available for connection appears. Select a signal or parameter from the list to connect the selected Dashboard block. To connect another Dashboard block, pause on the block and click the **Connect** button above it. Then, make a selection of signals and blocks in your model, and choose a signal or parameter to connect.
When you finish connecting the Dashboard blocks in your model, click the **Exit** button in the upper-right of the canvas to exit connect mode.

**Tip** You can hide the message shown on unconnected blocks using the `set_param` function and the `ShowInitialText` block parameter.

### Limitations

- Dashboard blocks can only connect to real scalar signals.
- You cannot use the **Connection** table to connect a Dashboard block to a block that is commented out. When you connect a Dashboard block to a commented block using connect mode, the Dashboard block does not display the connected value until you uncomment the block.
- Dashboard blocks cannot connect to signals inside referenced models.
- When you simulate a model hierarchy, Dashboard blocks inside referenced models do not update.
- Dashboard blocks do not support rapid accelerator simulation.
• Some signals are not available during accelerator mode simulations. To view such signals on a Dashboard block, mark the signal for logging.

Parameters

Connection — Select a signal to connect and display
signal connection options

Select the signal to connect using the Connection table. Populate the Connection table by selecting signals of interest in your model. Select the radio button next to the signal you want to display. Click Apply to connect the signal. To facilitate understanding and debugging your model, you can connect Dashboard blocks to signals in your model during simulation.

Programmatic Use

Block Parameter: Binding
Type: Simulink.HMI.SignalSpecification
Default: []

Minimum — Minimum tick mark value
0 (default) | scalar

Finite, real, double, scalar value specifying the minimum tick mark value for the arc. The minimum must be less than the value entered for the maximum.

Programmatic Use

To programmatically set the Minimum parameter, use a 1-by-3 vector containing values for the Minimum, Tick Interval, and Maximum parameters, in that order. To use the auto value for the Tick Interval, leave the Tick Interval position in the vector empty or specify -1.

Block Parameter: Limits
Type: 1x3 vector
Default: [0 -1 100]

Maximum — Maximum tick mark value
100 (default) | scalar

Finite, real, double, scalar value specifying the maximum tick mark value for the arc. The maximum must be greater than the value entered for the minimum.
Programmatic Use

To programmatically set the Maximum parameter, use a 1-by-3 vector containing values for the Minimum, Tick Interval, and Maximum parameters, in that order. To use the auto value for the Tick Interval, leave the Tick Interval position in the vector empty or specify -1.

**Block Parameter:** Limits
**Type:** 1x3 vector
**Default:** [0 -1 100]

**Tick Interval — Interval between major tick marks**
auto (default) | scalar

Finite, real, positive, integer, scalar value specifying the interval of major tick marks on the arc. When set to auto, the block automatically adjusts the tick interval based on the minimum and maximum values.

Programmatic Use

To programmatically set the Tick Interval parameter, use a 1-by-3 vector containing values for the Minimum, Tick Interval, and Maximum parameters, in that order. To use the auto value for the Tick Interval, leave the Tick Interval position in the vector empty or specify -1.

**Block Parameter:** Limits
**Type:** 1x3 vector
**Default:** [0 -1 100]

**Scale Colors — Color indications on gauge arc**
colors for arc ranges

Color specifications for ranges on the arc. Press the + button to add a color. For each color added, specify the minimum and maximum values of the range where you want to display that color.

Programmatic Use

To programmatically specify the Scale Colors parameter, use an array of structures with fields:

- Min — Minimum value for the color range on the scale.
- Max — Maximum value for the color range on the scale.
- Color — 1-by-3 vector of double values between 0 and 1 to specify the color for the range of the scale in the form [r g b]
Include a structure in the array for each scale range for which you want to specify a color.

```matlab
range1.Min = 0;
rang1.Max = 10;
rang1.Color = [0 0 1];
rang1.Min = 10;
rang1.Max = 15;
rang1.Color = [0 1 0];
scaleRanges = [rang1 rang2];
```

**Block Parameter:** ScaleColors  
**Type:** structure array  
**Default:** 0x1 struct array

**Label — Block label position**  
*Top (default) | Bottom | Hide*  

Position of the block label. When the block is connected to an element in the model, the label is the name of the connected element.

**Programmatic Use**  
**Block Parameter:** LabelPosition  
**Type:** character vector  
**Values:** 'Top' | 'Bottom' | 'Hide'  
**Default:** 'Top'

### Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>double</th>
<th>enumerated</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.
Dashboard blocks are ignored for code generation.

See Also
Custom Gauge | Gauge | Half Gauge | Quarter Gauge

Topics
“Tune and Visualize Your Model with Dashboard Blocks”
“Decide How to Visualize Simulation Data”

Introduced in R2015a
Logical Operator

Perform specified logical operation on input

Library: Simulink / Commonly Used Blocks
        Simulink / Logic and Bit Operations
        HDL Coder / Commonly Used Blocks
        HDL Coder / Logic and Bit Operations

Description

The Logical Operator block performs the specified logical operation on its inputs. An input value is true (1) if it is nonzero and false (0) if it is zero.

You select the Boolean operation connecting the inputs with the Operator parameter list. If you select rectangular as the Icon shape property, the name of the selected operator displays on the block icon. If you select distinctive as the Icon shape property, the name of the selected operator does not display on the block icon. This table shows supported operations:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND</td>
<td>TRUE if all inputs are TRUE</td>
</tr>
<tr>
<td>OR</td>
<td>TRUE if at least one input is TRUE</td>
</tr>
<tr>
<td>NAND</td>
<td>TRUE if at least one input is FALSE</td>
</tr>
<tr>
<td>NOR</td>
<td>TRUE when no inputs are TRUE</td>
</tr>
<tr>
<td>XOR</td>
<td>TRUE if an odd number of inputs are TRUE</td>
</tr>
<tr>
<td>NXOR</td>
<td>TRUE if an even number of inputs are TRUE</td>
</tr>
<tr>
<td>NOT</td>
<td>TRUE if the input is FALSE</td>
</tr>
</tbody>
</table>

If you select distinctive as the Icon shape, the block appearance indicates its function. Simulink software displays a distinctive shape for the selected operator, conforming to the IEEE Standard Graphic Symbols for Logic Functions.
To specify the number of input ports, use the **Number of input ports** parameter. The output type is specified using the **Output data type** parameter. An output value is 1 if TRUE and 0 if FALSE.

**Note** The output data type should represent zero exactly. Data types that satisfy this condition include signed and unsigned integers, and any floating-point data type.

The size of the output depends on input vector size and the selected operator:

- If the block has more than one input, any nonscalar inputs must have the same dimensions. For example, if any input is a 2-by-2 array, all other nonscalar inputs must also be 2-by-2 arrays.

  Scalar inputs are expanded to have the same dimensions as the nonscalar inputs.

  If the block has more than one input, the output has the same dimensions as the inputs (after scalar expansion) and each output element is the result of applying the specified logical operation to the corresponding input elements. For example, if the specified operation is AND and the inputs are 2-by-2 arrays, the output is a 2-by-2 array whose top left element is the result of applying AND to the top left elements of the inputs, and so on.

- For a single vector input, the block applies the operation (except the NOT operator) to all elements of the vector. The output is always a scalar.

- The NOT operator accepts only one input, which can be a scalar or a vector. If the input is a vector, the output is a vector of the same size containing the logical complements of the input vector elements.

When configured as a multi-input XOR gate, this block performs an addition modulo two operation as mandated by the IEEE Standard for Logic Elements.
Ports

Input

Port_1 — First input signal
scalar | vector | matrix

First input signal, specified as a scalar, vector, or matrix.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point

Port_2 — Second input signal
scalar | vector | matrix

Second input signal, specified as a scalar, vector, or matrix.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point

Port_N — Nth input signal
scalar | vector | matrix

Nth input signal, specified as a scalar, vector, or matrix.

Dependencies

To enable additional input ports, use the Number of input ports parameter.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point

Output

Port_1 — Output signal
scalar | vector | matrix

Output signal, consisting of zeros and ones, with the same dimensions as the input. You control the output data type with the Require all inputs and output to have the same data type and Output data type parameters.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point
Parameters

Main

**Operator — Logical operator**
AND (default) | OR | NAND | NOR | XOR | NXOR | NOT

Select the logical operator to apply to block inputs.

- **AND** — T if all inputs are TRUE
- **OR** — TRUE if at least one input is TRUE
- **NAND** — TRUE if at least one input is FALSE
- **NOR** — TRUE when no inputs are TRUE
- **XOR** — TRUE if an odd number of inputs are TRUE
- **NXOR** — TRUE if an even number of inputs are TRUE
- **XOR** — TRUE if the input is FALSE

**Programmatic Use**

**Block Parameter:** Operator
**Type:** character vector
**Values:** 'AND' | 'OR' | 'NAND' | 'NOR' | 'XOR' | 'NXOR' | 'NOT'
**Default:** 'AND'

**Number of input ports — Number of inputs**

2 (default) | positive integer

Specify the number of block inputs as a positive integer.

**Programmatic Use**

**Block Parameter:** Inputs
**Type:** character vector
**Values:** positive integer
**Default:** '2'

**Dependencies**

This parameter is not available when you set the **Operator** to NOT.

**Icon shape — Icon shape**

rectangular (default) | distinctive
Specify the shape of the block icon.

- **rectangular** — Results in a rectangular block that displays the name of the selected operator.
- **distinctive** — Use the graphic symbol for the selected operator as specified by the IEEE standard.

**Programmatic Use**

Block Parameter: IconShape

Type: character vector

Values: 'rectangular' | 'distinctive'

Default: 'rectangular'

**Sample time — Specify sample time as a value other than -1**

-1 (default) | scalar | vector

Specify the sample time as a value other than -1. For more information, see “Specify Sample Time”.

** Dependencies**

This parameter is not visible unless it is explicitly set to a value other than -1. To learn more, see “Blocks for Which Sample Time Is Not Recommended”.

**Programmatic Use**

Block Parameter: SampleTime

Type: character vector

Values: scalar or vector

Default: '-1'

**Data Type**

**Require all inputs and output to have the same data type — Require all ports to have same data type**

off (default) | on

To require that all block inputs and the output have the same data type, select this check box. When you clear this check box, the inputs and output can have different data types.

**Programmatic Use**

Block Parameter: AllPortsSameDT

Type: character vector
Values: 'off' | 'on
Default: 'off'

Output data type — Output data type
boolean (default) | Inherit: Logical (see Configuration Parameters: Optimization) | fixdt(1,16) | <data type expression>

Specify the output data type. When you select:

- boolean — The block output has data type boolean.
- Inherit: Logical (see Configuration Parameters: Optimization) — The block uses the Implement logic signals as Boolean data configuration parameter to specify the output data type (see “Implement logic signals as Boolean data (vs. double)”).

**Note** This option supports models created before the boolean option was available. Use one of the other options, preferably boolean, for new models.

- fixdt(1,16) — The block output has the specified fixed-point data type fixdt(1,16).

**Tip** Click the **Show data type assistant** button to display the Data Type Assistant, which helps you set the data type attributes. For more information, see “Specify Data Types Using Data Type Assistant”.

- <data type expression> — The block output has the data type you specify as a data type expression, for example, Simulink.NumericType.

**Tip** To enter a built-in data type (double, single, int8, uint8, int16, uint16, int32, or uint32), enclose the expression in single quotes. For example, enter 'double' instead of double.

Programmatic Use
**Block Parameter:** OutDataTypeStr
**Type:** character vector
**Values:** 'Inherit: Logical (see Configuration Parameters: Optimization)' | 'boolean' | 'fixdt(1,16)' | '<data type expression>'
Default: 'boolean'

## Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>double</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

## Extended Capabilities

### C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

### HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

### HDL Architecture
This block has a single, default HDL architecture.
HDL Block Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConstrainedOutputPipeline</td>
<td>Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. For more details, see “ConstrainedOutputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td>InputPipeline</td>
<td>Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “InputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td>OutputPipeline</td>
<td>Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “OutputPipeline” (HDL Coder).</td>
</tr>
</tbody>
</table>

PLC Code Generation
Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

See Also
“Implement logic signals as Boolean data (vs. double)” | Combinatorial Logic | Relational Operator

Introduced before R2006a
1-D Lookup Table

Approximate one-dimensional function

**Library:**
- Simulink / Lookup Tables
- HDL Coder / Lookup Tables

### Description

#### Supported Block Operations

The 1-D, 2-D, and n-D Lookup Table blocks evaluate a sampled representation of a function in N variables

\[ y = F(x_1, x_2, x_3, ..., x_N) \]

where the function \( F \) can be empirical. The block maps inputs to an output value by looking up or interpolating a table of values you define with block parameters. The block supports flat (constant), linear (linear point-slope), Lagrange (linear Lagrange), nearest, cubic-spline, and Akima spline interpolation methods. You can apply these methods to a table of any dimension from 1 through 30.

In the following block, the first input identifies the first dimension (row) breakpoints, the second input identifies the second dimension (column) breakpoints, and so on.

See “Port Location After Rotating or Flipping” for a description of the port order for various block orientations.
When the **Math and Data Types > Use algorithms optimized for row-major array layout** configuration parameter is set, the 2-D and n-D Lookup Table block behavior changes from column-major to row-major. For these blocks, the column-major and row-major algorithms may differ in the order of the output calculations, possibly resulting in slightly different numerical values. This capability requires a Simulink Coder or Embedded Coder license. For more information on row-major support, see “Code Generation of Matrices and Arrays” (Simulink Coder).

**Specification of Breakpoint and Table Data**

These block parameters define the breakpoint and table data.

<table>
<thead>
<tr>
<th>Block Parameter</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of table dimensions</td>
<td>Specifies the number of dimensions of your lookup table.</td>
</tr>
<tr>
<td>Breakpoints</td>
<td>Specifies a breakpoint vector that corresponds to each dimension of your lookup table.</td>
</tr>
<tr>
<td>Table data</td>
<td>Defines the associated set of output values.</td>
</tr>
</tbody>
</table>

**Tip** Evenly spaced breakpoints can make the generated code division-free. For more information, see `fixpt_evenspace_cleanup` and “Identify questionable fixed-point operations” (Embedded Coder).

**How the Block Generates Output**

The n-D, 1-D and 2-D Lookup Table blocks generate output by looking up or estimating table values based on the input values.

<table>
<thead>
<tr>
<th>Block Inputs</th>
<th>n-D Lookup Table Block Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>Match the values of indices in breakpoint data sets</td>
<td>Outputs the table value at the intersection of the row, column, and higher dimension breakpoints</td>
</tr>
<tr>
<td>Do not match the values of indices in breakpoint data sets, but are within range</td>
<td>Interpolates appropriate table values, using the <strong>Interpolation method</strong> you select</td>
</tr>
</tbody>
</table>
**Other Blocks that Perform Equivalent Operations**

You can use the Interpolation Using Prelookup block with the Prelookup block to perform the equivalent operation of one n-D Lookup Table block. This combination of blocks offers greater flexibility that can result in more efficient simulation performance for linear interpolations.

When the lookup operation is an array access that does not require interpolation, use the Direct Lookup Table (n-D) block. For example, if you have an integer value \( k \) and you want the \( k \)th element of a table, \( y = \text{table}(k) \), interpolation is unnecessary.

**Ports**

**Input**

**u1 — First-dimension (row) inputs**

scalar | vector | matrix

Real-valued inputs to the \( u1 \) port, mapped to an output value by looking up or interpolating the table of values that you define.

Example: 0:10

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | enumerated | fixed point

**Output**

**Port_1 — Output computed by looking up or estimating table values**

scalar | vector | matrix

Output generated by looking up or estimating table values based on the input values.
When block inputs... | The n-D Lookup Table block...
---|---
Match the values of indices in breakpoint data sets | Outputs the table value at the intersection of the row, column, and higher dimension breakpoints
Do not match the values of indices in breakpoint data sets, but are within range | Interpolates appropriate table values, using the Interpolation method you select
Do not match the values of indices in breakpoint data sets, and are out of range | Extrapolates the output value, using the Extrapolation method you select

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

## Parameters

### Table and Breakpoints

**Number of table dimensions — Number of lookup table dimensions**

1 (default) | 2 | 3 | 4 | ... | 30

Enter the number of dimensions of the lookup table. This parameter determines:
- The number of independent variables for the table and the number of block inputs
- The number of breakpoint sets to specify

<table>
<thead>
<tr>
<th>To specify...</th>
<th>Do this...</th>
</tr>
</thead>
<tbody>
<tr>
<td>1, 2, 3, or 4</td>
<td>Select the value from the drop-down list.</td>
</tr>
<tr>
<td>A higher number of table dimensions</td>
<td>Enter a positive integer directly in the field.</td>
</tr>
</tbody>
</table>

The maximum number of table dimensions that this block supports is 30.

For example, a table with a size of $M \times N \times \ldots$ means that the size of dimension 1 is $M$, the size of dimension 2 is $N$, and so forth. $M$ must match the first breakpoint length, $N$ must match the second breakpoint length, and so forth.

**Programmatic Use**

**Block Parameter:** `NumberOfTableDimensions`
**Type:** character vector

**Values:** '1' | '2' | '3' | '4' | ... | 30

**Default:** '1'

**Data specification — Method of table and breakpoint specification**

Table and breakpoints (default) | Lookup table object

From the list, select:

- **Table and breakpoints** — Specify the table data and breakpoints. Selecting this option enables these parameters:
  - **Table data**
  - **Breakpoints specification**
  - **Breakpoints 1**
  - **Edit table and breakpoints**
- **Lookup table object** — Use an existing lookup table (Simulink.LookupTable) object. Selecting this option enables the **Name** field and **Edit table and breakpoints** button.

**Programmatic Use**

**Block Parameter:** DataSpecification

**Type:** character vector

**Values:** 'Table and breakpoints' | 'Lookup table object'

**Default:** 'Table and breakpoints'

**Name — Name of the lookup table object**

[] (default) | Simulink.LookupTable object

Enter the name of the lookup table (Simulink.LookupTable) object. If a Simulink.LookupTable object does not exist, click the action button and select **Create**. The corresponding parameters of the new lookup table object are automatically populated with the block information.

**Dependencies**

To enable this parameter, set **Data specification** to **Lookup table object**.

**Programmatic Use**

**Block Parameter:** LookupTableObject
**Type:** character vector  
**Values:** name of a Simulink.LookupTable object  
**Default:** ''

**Table data — Define the table of output values**  
`tanh([-5:5])` (default) | vector of values

Enter the table of output values.

During simulation, the matrix size must match the dimensions defined by the **Number of table dimensions** parameter. However, during block diagram editing, you can enter an empty matrix (specified as `[]`) or an undefined workspace variable. This technique lets you postpone specifying a correctly dimensioned matrix for the table data and continue editing the block diagram.

**Dependencies**

To enable this parameter, set **Data specification** to Table and breakpoints.

**Programmatic Use**

**Block Parameter:** Table  
**Type:** character vector  
**Values:** vector of table values  
**Default:** `'tanh([-5:5])'`

**Breakpoints specification — Method of breakpoint specification**

Explicit values (default) | Even spacing

Specify whether to enter data as explicit breakpoints or as parameters that generate evenly spaced breakpoints.

- To explicitly specify breakpoint data, set this parameter to Explicit values and enter breakpoint data in the text box next to the Breakpoints parameters.
- To specify parameters that generate evenly spaced breakpoints, set this parameter to Even spacing and enter values for the First point and Spacing parameters for each dimension of breakpoint data. The block calculates the number of points to generate from the table data.

**Dependencies**

To enable this parameter, set **Data specification** to Table and breakpoints.
Programmatic Use
Block Parameter: BreakpointsSpecification
Type: character vector
Values: 'Explicit values' | 'Even spacing'
Default: 'Explicit values'

Breakpoints — Explicit breakpoint values, or first point and spacing of breakpoints
[10, 22, 31] (default) | 1-by-n or n-by-1 vector of monotonically increasing values

Specify the breakpoint data explicitly or as evenly-spaced breakpoints, based on the value of the Breakpoints specification parameter.

- If you set Breakpoints specification to Explicit values, enter the breakpoint set that corresponds to each dimension of table data in each Breakpoints row. For each dimension, specify breakpoints as a 1-by-n or n-by-1 vector whose values are strictly monotonically increasing.
- If you set Breakpoints specification to Even spacing, enter the parameters First point and Spacing in each Breakpoints row to generate evenly spaced breakpoints in the respective dimension. Your table data determines the number of evenly spaced points.

Dependencies
To enable this parameter, set Data specification to Table and breakpoints.

Programmatic Use
Block Parameter: BreakpointsForDimension1
Type: character vector
Values: 1-by-n or n-by-1 vector of monotonically increasing values
Default: '[10, 22, 31]'

First point — First point in evenly spaced breakpoint data
1 (default) | scalar

Specify the first point in your evenly spaced breakpoint data as a real-valued, finite, scalar. This parameter is available when Breakpoints specification is set to Even spacing.

Dependencies
To enable this parameter, set Data specification to Table and breakpoints, and Breakpoints specification to Even spacing.
Programmatic Use

Block Parameter: BreakpointsForDimension1FirstPoint
Type: character vector
Values: real-valued, finite, scalar
Default: '1'

Spacing — Spacing between evenly spaced breakpoints
1 (default) | scalar

Specify the spacing between points in your evenly spaced breakpoint data.

Dependencies

To enable this parameter, set Data specification to Table and breakpoints, and Breakpoints specification to Even spacing.

Programmatic Use

Block Parameter: BreakpointsForDimension1Spacing
Type: character vector
Values: positive, real-valued, finite, scalar
Default: '1'

Edit table and breakpoints — Launch Lookup Table Editor dialog box
button

Click this button to open the Lookup Table Editor. For more information, see “Edit Lookup Tables” in the Simulink documentation.

Clicking this button for a lookup table object lets you edit the object and save the new values for the object.

Algorithm

Lookup method

Interpolation method — Method of interpolation between breakpoint values
Linear point-slope (default) | Flat | Nearest | Linear Lagrange | Cubic spline | Akima spline

When an input falls between breakpoint values, the block interpolates the output value using neighboring breakpoints. For more information on interpolation methods, see “Interpolation Methods”.
Dependencies

- If you select **Cubic spline**, the block supports only scalar signals. The other interpolation methods support nonscalar signals.
- If you select **Akima spline**, the extrapolation method can only be **Akima spline**.

When set to the modified Akima interpolation method, this block does not support:

- Row-major array layout and algorithms optimized for row-major array layout
- Scaled double and fixed-point data types
- Simulink.LookupTable objects
- Code generation when the configuration parameter **Code Generation > Interface > Support non-finite numbers** check box is selected

Programmatic Use

**Block Parameter:** InterpMethod  
**Type:** character vector  
**Values:** 'Linear point-slope' | 'Flat' | 'Nearest' | 'Linear Lagrange' | 'Cubic spline' | 'Akima spline'  
**Default:** 'Linear point-slope'

Extrapolation method — Method of handling input values that fall outside the range of a breakpoint data set

Clip (default) | Linear | Cubic spline | Akima spline

Select Clip, Linear, or Cubic spline. See “Extrapolation Methods” for more information.

If the extrapolation method is Linear, the extrapolation value is calculated based on the selected linear interpolation method. For example, if the interpolation method is linear Lagrange, the extrapolation method inherits the linear Lagrange equation to compute the extrapolated value.

Dependencies

- To select **Cubic spline** for **Extrapolation method**, you must also select **Cubic spline** for **Interpolation method**.
- To select **Akima spline** for **Extrapolation method**, you must also select **Akima spline** for **Interpolation method**.
Programmatic Use

Block Parameter: ExtrapMethod
Type: character vector
Values: 'Linear' | 'Clip' | 'Cubic spline' | 'Akima spline'
Default: 'Linear'

Index search method — Method of calculating table indices

Evenly spaced points (default) | Linear search | Binary search

Select Evenly spaced points, Linear search, or Binary search. Each search method has speed advantages in different circumstances:

- For evenly spaced breakpoint sets (for example, 10, 20, 30, and so on), you achieve optimal speed by selecting Evenly spaced points to calculate table indices. This algorithm uses only the first two breakpoints of a set to determine the offset and spacing of the remaining points.

  **Note** Set Index search method to Evenly spaced points when using the Simulink.LookupTable object to specify table data and the Breakpoints Specification parameter of the referenced Simulink.LookupTable object is set to Even spacing.

- For unevenly spaced breakpoint sets, follow these guidelines:
  - If input signals do not vary much between time steps, selecting Linear search with Begin index search using previous index result produces the best performance.
  - If input signals jump more than one or two table intervals per time step, selecting Binary search produces the best performance.

A suboptimal choice of index search method can lead to slow performance of models that rely heavily on lookup tables.

**Note** The generated code stores only the first breakpoint, the spacing, and the number of breakpoints when:

- The breakpoint data is not tunable.
- The index search method is Evenly spaced points.
Programmatic Use
Block Parameter: IndexSearchMethod
Type: character vector
Values: 'Binary search' | 'Evenly spaced points' | 'Linear search'
Default: 'Binary search'

Begin index search using previous index result — Start using the index from the previous time step
off (default) | on

Select this check box when you want the block to start its search using the index found at the previous time step. For inputs that change slowly with respect to the interval size, enabling this option can improve performance. Otherwise, the linear search and binary search methods can take longer, especially for large breakpoint sets.

Dependencies
To enable this parameter, set Index search method to Linear search or Binary search.

Programmatic Use
Block Parameter: BeginIndexSearchUsingPreviousIndexResult
Type: character vector
Values: 'off' | 'on'
Default: 'off'

Diagnostic for out-of-range input — Block action when input is out of range
None (default) | Warning | Error

Specify whether to produce a warning or error when the input is out of range. Options include:

• None — Produce no response.
• Warning — Display a warning and continue the simulation.
• Error — Terminate the simulation and display an error.

Programmatic Use
Block Parameter: DiagnosticForOutOfRangeInput
Type: character vector
Values: 'None' | 'Warning' | 'Error'
Default: 'None'
Use last table value for inputs at or above last breakpoint — Method for computing output for inputs at or above last breakpoint

off (default) | on

Using this check box, specify the indexing convention that the block uses to address the last element of a breakpoint set and its corresponding table value. This check box is relevant if the input is equal to or larger than the last element of the breakpoint data. Due to rounding, selecting and clearing this check box may result in differing results for the last breakpoint between simulation and code generation.

<table>
<thead>
<tr>
<th>Check Box</th>
<th>Index Used by Block</th>
<th>Interval Fraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selected</td>
<td>Last element of breakpoint data on the Table and Breakpoints tab</td>
<td>0</td>
</tr>
<tr>
<td>Cleared</td>
<td>Next-to-last element of breakpoint data on the Table and Breakpoints tab</td>
<td>1</td>
</tr>
</tbody>
</table>

Given an input $u$ within range of a breakpoint set $bp$, the interval fraction $f$, in the range $0 \leq f \leq 1$, is computed as shown below.

\[
f = \frac{u - bp(n)}{bp(n+1) - bp(n)} = \frac{a}{b}
\]

Suppose the breakpoint set is $[1 \ 4 \ 5]$ and input $u$ is $5.5$. If you select this check box, the index is that of the last element (5) and the interval fraction is 0. If you clear this check box, the index is that of the next-to-last element (4) and the interval fraction is 1.
**Dependencies**

To enable this parameter, set:

- **Interpolation method** to **Linear**.
- **Extrapolation method** to **Clip**.

**Programmatic Use**

**Block Parameter:** `UseLastTableValue`

- **Type:** character vector
- **Values:** 'off' | 'on'
- **Default:** 'off'

**Input settings**

**Use one input port for all input data — Use only one input port**

- **off** (default) | **on**

Select this check box to use only one input port that expects a signal that is \( n \) elements wide for an \( n \)-dimensional table. This option is useful for removing line clutter on a block diagram with many lookup tables.

**Note** When you select this check box, one input port with the label \( u \) appears on the block.

**Programmatic Use**

**Block Parameter:** `UseOneInputPortForAllInputData`

- **Type:** character vector
- **Values:** 'off' | 'on'
- **Default:** 'off'

**Code generation**

**Remove protection against out-of-range input in generated code — Remove code that checks for out-of-range input values**

- **off** (default) | **on**

Specify whether or not to include code that checks for out-of-range input values.
<table>
<thead>
<tr>
<th>Check Box</th>
<th>Result</th>
<th>When to Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>on</td>
<td>Generated code does not include conditional statements to check for out-of-range breakpoint inputs. When the input is out-of-range, it may cause undefined behavior for generated code and simulations using accelerator mode.</td>
<td>For code efficiency</td>
</tr>
<tr>
<td>off</td>
<td>Generated code includes conditional statements to check for out-of-range inputs.</td>
<td>For safety-critical applications</td>
</tr>
</tbody>
</table>

If your input is not out of range, you can select the **Remove protection against out-of-range index in generated code** check box for code efficiency. By default, this check box is cleared. For safety-critical applications, do not select this check box. If you want to select the **Remove protection against out-of-range index in generated code** check box, first check that your model inputs are in range. For example:

1. Clear the **Remove protection against out-of-range index in generated code** check box.
2. Set the **Diagnostic for out-of-range input** parameter to **Error**.
3. Simulate the model in normal mode.
4. If there are out-of-range errors, fix them to be in range and run the simulation again.
5. When the simulation no longer generates out-of-range input errors, select the **Remove protection against out-of-range index in generated code** check box.

**Note** When you select the **Remove protection against out-of-range index in generated code** check box and the input k or f is out of range, the behavior is undefined for generated code and simulations using accelerator mode.

Depending on your application, you can run the following Model Advisor checks to verify the usage of this check box:
• **By Product > Embedded Coder >** Identify lookup table blocks that generate expensive out-of-range checking code

• **By Product > Simulink Check > Modeling Standards > DO-178C/DO-331 Checks >** Check usage of lookup table blocks

For more information about the Model Advisor, see “Run Model Advisor Checks”.

**Programmatic Use**

**Block Parameter:** RemoveProtectionInput  
**Type:** character vector  
**Values:** 'off' | 'on'  
**Default:** 'off'

**Support tunable table size in code generation — Enable tunable table size in the generated code**  
off (default) | on

Select this check box to enable tunable table size in the generated code. This option enables you to change the size and values of the lookup table and breakpoint data in the generated code without regenerating or recompiling the code. You can only decrease the size of the lookup table and breakpoint data.

**Dependencies**

If you set **Interpolation method** to Cubic spline, this check box is not available.

**Programmatic Use**

**Block Parameter:** SupportTunableTableSize  
**Type:** character vector  
**Values:** 'off' | 'on'  
**Default:** 'off'

**Sample time — Specify sample time as a value other than -1**  
-1 (default) | scalar | vector

Specify the sample time as a value other than -1. For more information, see “Specify Sample Time”.

**Dependencies**

This parameter is not visible unless it is explicitly set to a value other than -1. To learn more, see “Blocks for Which Sample Time Is Not Recommended”.

1-1131
**Programmatic Use**

**Block Parameter:** SampleTime  
**Type:** character vector  
**Values:** scalar or vector  
**Default:** '-1'

**Maximum indices for each dimension — Maximum index value for each table dimension**

[] (default) | scalar or vector of positive integer values

Example: [4 6] for a 5-by-7 table

Specify the maximum index values for each table dimension using zero-based indexing. You can specify a scalar or vector of positive integer values using the following data types:

- Built-in floating-point types: double and single
- Built-in integer types: int8, int16, int32, uint8, uint16, and uint32

Examples of valid specifications include:

- [4 6] for a 5-by-7 table
- [int8(2) int16(5) int32(9)] for a 3-by-6-by-10 table
- A Simulink.Parameter whose value on generating code is one less than the dimensions of the table data. For more information, see “Tunable Table Size in the Generated Code” on page 1-1142.

**Dependencies**

To enable this parameter, select **Support tunable table size in code generation**. On tuning this parameter in the generated code, provide the new table data and breakpoints along with the tuned parameter value.

**Programmatic Use**

**Block Parameter:** MaximumIndicesForEachDimension  
**Type:** character vector  
**Values:** scalar or vector of positive integer values  
**Default:** '[ ]'
Data Types

**Table data — Data type of table data**

*Inherit: Same as output* (default) | double | single | int8 | uint8 | int16 | uint16 | int32 | uint32 | int64 | uint64 | fixdt(1,16) | fixdt(1,16,0) | fixdt(1,16,2^0,0) | <data type expression>

Specify the table data type. You can set it to:

- A rule that inherits a data type, for example, Inherit: Same as output
- The name of a built-in data type, for example, single
- The name of a data type object, for example, a Simulink.NumericType object
- An expression that evaluates to a data type, for example, fixdt(1,16,0)

Click the **Show data type assistant** button to display the **Data Type Assistant**, which helps you set the data type attributes. For more information, see “Specify Data Types Using Data Type Assistant”.

**Tip** Specify a table data type different from the output data type for these cases:

- Lower memory requirement for storing table data that uses a smaller type than the output signal
- Sharing of prescaled table data between two n-D Lookup Table blocks with different output data types
- Sharing of custom storage table data in the generated code for blocks with different output data types

**Programmatic Use**

**Block Parameter**: TableDataTypeStr  
**Type**: character vector  
**Values**: 'Inherit: Inherit from 'Table data'' | 'Inherit: Same as output' | 'double' | 'single' | 'int8' | 'uint8' | 'int16' | 'uint16' | 'int32' | 'uint32' | 'int64' | 'uint64' | 'fixdt(1,16)' | 'fixdt(1,16,0)' | 'fixdt(1,16,2^0,0)' | '<data type expression>'  
**Default**: 'Inherit: Same as output'
Table data Minimum — Minimum value of the table data
[] | scalar

Specify the minimum value for table data. The default value is [] (unspecified).

Programmatic Use
Block Parameter: TableMin
Type: character vector
Values: scalar
Default: '[]'

Table data Maximum — Maximum value of the table data
[] | scalar

Specify the maximum value for table data. The default value is [] (unspecified).

Programmatic Use
Block Parameter: TableMax
Type: character vector
Values: scalar
Default: '[]'

Breakpoints — Breakpoint data type
Inherit: Same as corresponding input (default) | double | single | int8 | uint8 | int16 | uint16 | int32 | uint32 | int64 | uint64 | fixdt(1,16) | fixdt(1,16,0) | fixdt(1,16,2^0,0) | Enum: <class name> | <data type expression>

Specify the data type for a set of breakpoint data. You can set it to:

- A rule that inherits a data type, for example, Inherit: Same as corresponding input
- The name of a built-in data type, for example, single
- The name of a data type class, for example, an enumerated data type class
- The name of a data type object, for example, a Simulink.NumericType object
- An expression that evaluates to a data type, for example, fixdt(1,16,0)

Tip
• Breakpoints support unordered enumerated data. As a result, linear searches are also unordered, which offers flexibility but can impact performance. The search begins from the first element in the breakpoint.
• If the **Begin index search using previous index result** check box is selected, you must use ordered monotonically increasing data. This ordering improves performance.
• For enumerated data, **Extrapolation method** must be **Clip**.
• The block does not support out-of-range input for enumerated data. When specifying enumerated data, include the entire enumeration set in the breakpoint data set. For example, use the `enumeration` function.

This is a limitation for using enumerated data with this block:

• The block does not support out-of-range input for enumerated data. When specifying enumerated data, include the entire enumeration set in the breakpoint data set. For example, use the `enumeration` function.

Click the **Show data type assistant** button to display the **Data Type Assistant**, which helps you set the data type attributes. For more information, see “Specify Data Types Using Data Type Assistant”.

**Tip** Specify a breakpoint data type different from the corresponding input data type for these cases:

• Lower memory requirement for storing breakpoint data that uses a smaller type than the input signal
• Sharing of prescaled breakpoint data between two n-D Lookup Table blocks with different input data types
• Sharing of custom storage breakpoint data in the generated code for blocks with different input data types

**Programmatic Use**
**Block Parameter:** BreakpointsForDimension1DataTypeStr | BreakpointsForDimension2DataTypeStr | ... | BreakpointsForDimension30DataTypeStr
**Type:** character vector  
**Values:** 'Inherit: Same as corresponding input' | 'Inherit: Inherit from 'Breakpoint data' | 'double' | 'single' | 'int8' | 'uint8' | 'int16' | 'uint16' | 'int32' | 'uint32' | 'int64' | 'uint64' | 'fixdt(1,16)' | 'fixdt(1,16,0)' | 'fixdt(1,16,2^0,0)'</expression>  
**Default:** 'Inherit: Same as corresponding input'

**Breakpoints Minimum — Minimum value breakpoint data can have**  
[] | scalar

Specify the minimum value that a set of breakpoint data can have. The default value is [] (unspecified).

**Programmatic Use**  
**Block Parameter:** BreakpointsForDimension1Min | BreakpointsForDimension2Min | ... | BreakpointsForDimension30Min  
**Type:** character vector  
**Values:** scalar  
**Default:** '[]'

**Breakpoints Maximum — Maximum value breakpoint data can have**  
[] | scalar

Specify the maximum value that a set of breakpoint data can have. The default value is [] (unspecified).

**Programmatic Use**  
**Block Parameter:** BreakpointsForDimension1Max | BreakpointsForDimension2Max | ... | BreakpointsForDimension30Max  
**Type:** character vector  
**Values:** scalar  
**Default:** '[]'

**Fraction — Fraction data type**  
Inherit: Inherit via internal rule (default) | double | single | fixdt(1,16,0) | <data type expression>

Specify the fraction data type. You can set it to:

- A rule that inherits a data type, for example, Inherit: Inherit via internal rule
• The name of a built-in data type, for example, single
• The name of a data type object, for example, a Simulink.NumericType object
• An expression that evaluates to a data type, for example, fixdt(1,16,0)

Click the **Show data type assistant** button [>>] to display the **Data Type Assistant**, which helps you set the data type attributes. For more information, see “Specify Data Types Using Data Type Assistant”.

**Programmatic Use**

**Block Parameter:** FractionDataTypeStr  
**Type:** character vector  
**Values:** 'Inherit: Inherit via internal rule' | 'double' | 'single' | 'fixdt(1,16,0)' | '<data type expression>'  
**Default:** 'Inherit: Inherit via internal rule'

**Intermediate results — Intermediate results data type**

Inherit: Same as output (default) | double | single | int8 | uint8 | int16 | uint16 | int32 | uint32 | int64 | uint64 | fixdt(1,16) | fixdt(1,16,0) | fixdt(1,16,2^0,0) | '<data type expression>'

Specify the intermediate results data type. You can set it to:

• A rule that inherits a data type, for example, Inherit: Same as output
• The name of a built-in data type, for example, single
• The name of a data type object, for example, a Simulink.NumericType object
• An expression that evaluates to a data type, for example, fixdt(1,16,0)

Click the **Show data type assistant** button [>>] to display the **Data Type Assistant**, which helps you set the data type attributes. For more information, see “Specify Data Types Using Data Type Assistant”.

**Tip** Use this parameter to specify higher (or lower) precision for internal computations than for table data or output data.

**Programmatic Use**

**Block Parameter:** IntermediateResultsDataTypeStr
Type: character vector
Values: 'Inherit: Inherit via internal rule' | 'Inherit: Same as output' | 'double' | 'single' | 'int8' | 'uint8' | 'int16' | 'uint16' | 'int32' | 'uint32' | 'int64' | 'uint64' | 'fixdt(1,16,0)' | 'fixdt(1,16,2^0,0)' | '<data type expression>
Default: 'Inherit: Same as output'

Output — Output data type
Inherit: Same as input (default) | double | single | int8 | uint8 | int16 | uint16 | int32 | uint32 | int64 | uint64 | fixdt(1,16) | fixdt(1,16,0) | fixdt(1,16,2^0,0) | '<data type expression'>

Specify the output data type. You can set it to:

- A rule that inherits a data type, for example, Inherit: Inherit via back propagation
- The name of a built-in data type, for example, single
- The name of a data type object, for example, a Simulink.NumericType object
- An expression that evaluates to a data type, for example, fixdt(1,16,0)

Click the Show data type assistant button to display the Data Type Assistant, which helps you set the data type attributes. For more information, see “Specify Data Types Using Data Type Assistant”.

Programmatic Use
Block Parameter: OutDataTypeStr
Type: character vector
Values: 'Inherit: Inherit via back propagation' | 'Inherit: Inherit from table data' | 'Inherit: Same as first input' | 'double' | 'single' | 'int8' | 'uint8' | 'int16' | 'uint16' | 'int32' | 'uint32' | 'int64' | 'uint64' | 'fixdt(1,16,0)' | 'fixdt(1,16,2^0,0)' | '<data type expression>
Default: 'Inherit: Same as first input'

Output Minimum — Minimum value the block can output
[] | scalar

Specify the minimum value that the block outputs. The default value is [] (unspecified). Simulink software uses this value to perform:
• Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters”).

• Simulation range checking (see “Signal Ranges”).

• Automatic scaling of fixed-point data types.

• Optimization of the code that you generate from the model. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. For more information, see “Optimize using the specified minimum and maximum values” (Simulink Coder).

Programmatic Use
Block Parameter: OutMin
Type: character vector
Values: scalar
Default: '[]'

Output Maximum — Maximum value the block can output
[] | scalar

Specify the maximum value that the block can output. The default value is [] (unspecified). Simulink software uses this value to perform:

• Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters”).

• Simulation range checking (see “Signal Ranges”).

• Automatic scaling of fixed-point data types.

• Optimization of the code that you generate from the model. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. For more information, see “Optimize using the specified minimum and maximum values” (Simulink Coder).

Programmatic Use
Block Parameter: OutMax
Type: character vector
Values: scalar
Default: '[]'

Internal rule priority — Internal rule for intermediate calculations
Speed (default) | Precision
Specify the internal rule for intermediate calculations. Select Speed for faster calculations. If you do, a loss of accuracy might occur, usually up to 2 bits.

**Programmatic Use**

**Block Parameter:** InternalRulePriority  
**Type:** character vector  
**Values:** 'Speed' | 'Precision'  
**Default:** 'Speed'

**Require all inputs to have the same data type — Require all inputs to have the same data type**

on (default) | off

Select to require all inputs to have the same data type.

**Programmatic Use**

**Block Parameter:** InputSameDT  
**Type:** character vector  
**Values:** 'off' | 'on'  
**Default:** 'on'

**Lock data type settings against changes by the fixed-point tools — Prevent fixed-point tools from overriding data types**

off (default) | on

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on this block. For more information, see “Lock the Output Data Type Setting” (Fixed-Point Designer).

**Programmatic Use**

**Block Parameter:** LockScale  
**Type:** character vector  
**Values:** 'off' | 'on'  
**Default:** 'off'

**Integer rounding mode — Rounding mode for fixed-point operations**

Simplest (default) | Ceiling | Convergent | Floor | Nearest | Round | Zero

Specify the rounding mode for fixed-point lookup table calculations that occur during simulation or execution of code generated from the model. For more information, see “Rounding” (Fixed-Point Designer).

This option does not affect rounding of values of block parameters. Simulink rounds such values to the nearest representable integer value. To control the rounding of a block
parameter, enter an expression using a MATLAB rounding function into the edit field on the block dialog box.

**Programmatic Use**

- **Block Parameter:** RndMeth
- **Type:** character vector
- **Values:** 'Ceiling' | 'Convergent' | 'Floor' | 'Nearest' | 'Round' | 'Simplest' | 'Zero'
- **Default:** 'Simplest'

**Saturate on integer overflow — Method of overflow action**

off (default) | on

<table>
<thead>
<tr>
<th>Action</th>
<th>Reasons for Taking This Action</th>
<th>What Happens for Overflows</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Select this check box (on).</td>
<td>Your model has possible overflow and you want explicit saturation protection in the generated code.</td>
<td>Overflows saturate to either the minimum or maximum value that the data type can represent.</td>
<td>An overflow associated with a signed 8-bit integer can saturate to -128 or 127.</td>
</tr>
<tr>
<td>Do not select this check box (off).</td>
<td>You want to optimize efficiency of your generated code. You want to avoid overspecifying how a block handles out-of-range signals. For more information, see “Check for Signal Range Errors”.</td>
<td>Overflows wrap to the appropriate value that is representable by the data type.</td>
<td>The number 130 does not fit in a signed 8-bit integer and wraps to -126.</td>
</tr>
</tbody>
</table>

**Tip** If you save your model as version R2009a or earlier, this check box setting has no effect and no saturation code appears. This behavior preserves backward compatibility.

When you select this check box, saturation applies to every internal operation on the block, not just the output or result. In general, the code generation process can detect when overflow is not possible. In this case, the code generator does not produce saturation code.
Programmatic Use
Block Parameter: SaturateOnIntegerOverflow
Type: character vector
Values: 'off' | 'on'
Default: 'off'

Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>double</th>
<th>enumerated</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

More About

Tunable Table Size in the Generated Code

Suppose that you have a lookup table and want to make the size tunable in the generated code. When you use Simulink.LookupTable and Simulink.Breakpoint objects to configure lookup table data for calibration in the generated code, use the SupportTunableSize property of the objects to enable a tunable table size. When you do not use these classes, use the Support tunable table size in code generation parameter in an n-D Lookup Table block to enable a tunable table size.

Assume that:

- You define a Simulink.Parameter structure in the preload function of your model:

```matlab
p = Simulink.Parameter;
p.Value.MaxIdx = [2 2];
```
p.Value.BP1 = [1 2 3];
p.Value.BP2 = [1 4 16];
p.Value.Table = [4 5 6; 16 19 20; 10 18 23];
p.DataType = 'Bus: slLookupTable';
p.CoderInfo.StorageClass = 'ExportedGlobal';

% Create bus object slBus1 from MATLAB structure
Simulink.Bus.createObject(p.Value);
slLookupTable = slBus1;
slLookupTable.Elements(1).DataType = 'uint32';

• These block parameters apply in the n-D Lookup Table block.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of table dimensions</td>
<td>2</td>
</tr>
<tr>
<td>Table data</td>
<td>p.Table</td>
</tr>
<tr>
<td>Breakpoints 1</td>
<td>p.BP1</td>
</tr>
<tr>
<td>Breakpoints 2</td>
<td>p.BP2</td>
</tr>
<tr>
<td>Support tunable table size in code generation</td>
<td>on</td>
</tr>
<tr>
<td>Maximum indices for each dimension</td>
<td>p.MaxIdx</td>
</tr>
</tbody>
</table>

The generated `model_types.h` header file contains a type definition that looks something like this.

```c
typedef struct {
    uint32_T MaxIdx[2];
    real_T BP1[3];
    real_T BP2[3];
    real_T Table[9];
} slLookupTable;
```

The generated `model.c` file contains code that looks something like this.

```c
/* Exported block parameters */
slLookupTable p = {
    { 2U, 2U },
    { 1.0, 2.0, 3.0 },
    { 1.0, 4.0, 16.0 },
```
The highlighted line of code specifies a tunable table size for the lookup table. You can change the size and values of the lookup table and breakpoint data without regenerating or recompiling the code.

**Enumerated Values in Lookup Tables**

Suppose that you have a lookup table with an enumerated class like this defined:

```matlab
classdef(Enumeration) Gears < Simulink.IntEnumType
    enumeration
        GEAR1(1),
        GEAR2(2),
        GEAR3(4),
        GEAR4(8),
        SPORTS(16),
        REVERSE(-1),
        NEUTRAL(0)
    end
end
```
n-D Lookup block has these settings:

- **Number of dimensions** to 1.
- **Table data** value is \([5 \ 10 \ 20 \ 40 \ 80 \ -5 \ 0]\).
- **Breakpoints 1** value is enumeration('Gears').
- Interpolation method is Flat.
- For an unordered search, set **Index search method** to Linear search and clear the **Begin index search using previous index result** check box.

Simulation produces a vector \([10 \ -5 \ 80]\), which correspond to GEAR2, REVERSE, and SPORTS.

![Diagram](image)

**Extended Capabilities**

**C/C++ Code Generation**
Generate C and C++ code using Simulink® Coder™.

**HDL Code Generation**
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic. For information about HDL code generation support for 1-D Lookup Table blocks, see “HDL Code Generation” on page 1-1205.

**Fixed-Point Conversion**
Design and simulate fixed-point systems using Fixed-Point Designer™.
See Also
Direct Lookup Table (n-D) | Interpolation Using Prelookup | Lookup Table Dynamic | Prelookup | Simulink.Breakpoint | Simulink.LookupTable

Topics
“Import Lookup Table Data from MATLAB”
“About Lookup Table Blocks”
“Anatomy of a Lookup Table”
“Enter Breakpoints and Table Data”
“Guidelines for Choosing a Lookup Table”

Introduced in R2011a
2-D Lookup Table

Approximate two-dimensional function

Library: Simulink / Lookup Tables
        HDL Coder / Lookup Tables

Description

Supported Block Operations

The 1-D, 2-D, and n-D Lookup Table blocks evaluate a sampled representation of a function in N variables

\[ y = F(x_1, x_2, x_3, ..., x_N) \]

where the function \( F \) can be empirical. The block maps inputs to an output value by looking up or interpolating a table of values you define with block parameters. The block supports flat (constant), linear (linear point-slope), Lagrange (linear Lagrange), nearest, cubic-spline, and Akima spline interpolation methods. You can apply these methods to a table of any dimension from 1 through 30.

In the following block, the first input identifies the first dimension (row) breakpoints, the second input identifies the second dimension (column) breakpoints, and so on.
See “Port Location After Rotating or Flipping” for a description of the port order for various block orientations.

When the Math and Data Types > Use algorithms optimized for row-major array layout configuration parameter is set, the 2-D and n-D Lookup Table block behavior changes from column-major to row-major. For these blocks, the column-major and row-major algorithms may differ in the order of the output calculations, possibly resulting in slightly different numerical values. This capability requires a Simulink Coder or Embedded Coder license. For more information on row-major support, see “Code Generation of Matrices and Arrays” (Simulink Coder).

**Specification of Breakpoint and Table Data**

These block parameters define the breakpoint and table data.

<table>
<thead>
<tr>
<th>Block Parameter</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Number of table dimensions</strong></td>
<td>Specifies the number of dimensions of your lookup table.</td>
</tr>
<tr>
<td><strong>Breakpoints</strong></td>
<td>Specifies a breakpoint vector that corresponds to each dimension of your lookup table.</td>
</tr>
<tr>
<td><strong>Table data</strong></td>
<td>Defines the associated set of output values.</td>
</tr>
</tbody>
</table>

**Tip** Evenly spaced breakpoints can make the generated code division-free. For more information, see fixpt_evenspace_cleanup and “Identify questionable fixed-point operations” (Embedded Coder).

**How the Block Generates Output**

The n-D, 1-D and 2-D Lookup Table blocks generate output by looking up or estimating table values based on the input values.

<table>
<thead>
<tr>
<th>Block Inputs</th>
<th>n-D Lookup Table Block Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>Match the values of indices in breakpoint data sets</td>
<td>Outputs the table value at the intersection of the row, column, and higher dimension breakpoints</td>
</tr>
</tbody>
</table>
**Block Inputs** | **n-D Lookup Table Block Behavior**
--- | ---
Do not match the values of indices in breakpoint data sets, but are within range | Interpolates appropriate table values, using the **Interpolation method** you select
Do not match the values of indices in breakpoint data sets, and are out of range | Extrapolates the output value, using the **Extrapolation method** you select

**Other Blocks that Perform Equivalent Operations**

You can use the Interpolation Using Prelookup block with the Prelookup block to perform the equivalent operation of one n-D Lookup Table block. This combination of blocks offers greater flexibility that can result in more efficient simulation performance for linear interpolations.

When the lookup operation is an array access that does not require interpolation, use the Direct Lookup Table (n-D) block. For example, if you have an integer value \(k\) and you want the \(k\)th element of a table, \(y = \text{table}(k)\), interpolation is unnecessary.

**Ports**

**Input**

**u1 — First-dimension (row) inputs**

scalar | vector | matrix

Real-valued inputs to the **u1** port, mapped to an output value by looking up or interpolating the table of values that you define.

Example: \(0:10\)

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | enumerated | fixed point

**u2 — Second-dimension (column) inputs**

scalar | vector | matrix

Real-valued inputs to the **u2** port, mapped to an output value by looking up or interpolating the table of values that you define.

Example: \(0:10\)
Output

Port_1 — Output computed by looking up or estimating table values
scalar | vector | matrix

Output generated by looking up or estimating table values based on the input values.

<table>
<thead>
<tr>
<th>When block inputs...</th>
<th>The n-D Lookup Table block...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Match the values of indices in breakpoint data sets</td>
<td>Outputs the table value at the intersection of the row, column, and higher dimension breakpoints</td>
</tr>
<tr>
<td>Do not match the values of indices in breakpoint data sets, but are within range</td>
<td>Interpolates appropriate table values, using the Interpolation method you select</td>
</tr>
<tr>
<td>Do not match the values of indices in breakpoint data sets, and are out of range</td>
<td>Extrapolates the output value, using the Extrapolation method you select</td>
</tr>
</tbody>
</table>

Parameters

Table and Breakpoints

Number of table dimensions — Number of lookup table dimensions
2 (default) | 1 | 3 | 4 | ... | 30

Enter the number of dimensions of the lookup table. This parameter determines:

- The number of independent variables for the table and the number of block inputs
- The number of breakpoint sets to specify

To specify... | Do this...
--- | ---
1, 2, 3, or 4 | Select the value from the drop-down list.
To specify... | Do this...
---|---
A higher number of table dimensions | Enter a positive integer directly in the field. The maximum number of table dimensions that this block supports is 30.

For example, a table with a size of $M \times N \times \ldots$ means that the size of dimension 1 is $M$, the size of dimension 2 is $N$, and so forth. $M$ must match the first breakpoint length, $N$ must match the second breakpoint length, and so forth.

**Programmatic Use**

**Block Parameter:** NumberOfTableDimensions  
**Type:** character vector  
**Values:** '1' | '2' | '3' | '4' | ... | 30  
**Default:** '2'

**Data specification — Method of table and breakpoint specification**

Table and breakpoints (default) | Lookup table object

From the list, select:

- **Table and breakpoints** — Specify the table data and breakpoints. Selecting this option enables these parameters:
  - **Table data**
  - **Breakpoints specification**
  - **Breakpoints 1**
  - **Breakpoints 2**
  - **Edit table and breakpoints**

- **Lookup table object** — Use an existing lookup table (Simulink.LookupTable) object. Selecting this option enables the **Name** field and **Edit table and breakpoints** button.

**Programmatic Use**

**Block Parameter:** DataSpecification  
**Type:** character vector  
**Values:** 'Table and breakpoints' | 'Lookup table object'  
**Default:** 'Table and breakpoints'
Name — Name of the lookup table object

Enter the name of the lookup table (Simulink.LookupTable) object. If a Simulink.LookupTable object does not exist, click the action button and select Create. The corresponding parameters of the new lookup table object are automatically populated with the block information.

Dependencies

To enable this parameter, set Data specification to Lookup table object.

Programmatic Use

Block Parameter: LookupTableObject
Type: character vector
Values: name of a Simulink.LookupTable object
Default: ''

Table data — Define the table of output values

Enter the table of output values.

During simulation, the matrix size must match the dimensions defined by the Number of table dimensions parameter. However, during block diagram editing, you can enter an empty matrix (specified as []) or an undefined workspace variable. This technique lets you postpone specifying a correctly dimensioned matrix for the table data and continue editing the block diagram.

Dependencies

To enable this parameter, set Data specification to Table and breakpoints.

Programmatic Use

Block Parameter: Table
Type: character vector
Values: matrix of table values
Default: '[4 5 6; 16 19 20; 10 18 23]'
Specify whether to enter data as explicit breakpoints or as parameters that generate evenly spaced breakpoints.

- To explicitly specify breakpoint data, set this parameter to Explicit values and enter breakpoint data in the text box next to the Breakpoints parameters.
- To specify parameters that generate evenly spaced breakpoints, set this parameter to Even spacing and enter values for the First point and Spacing parameters for each dimension of breakpoint data. The block calculates the number of points to generate from the table data.

**Dependencies**

To enable this parameter, set Data specification to Table and breakpoints.

**Programmatic Use**

**Block Parameter:** BreakpointsSpecification  
**Type:** character vector  
**Values:** 'Explicit values' | 'Even spacing'  
**Default:** 'Explicit values'

**Breakpoints — Explicit breakpoint values, or first point and spacing of breakpoints**

[1:3] (default) | 1-by-n or n-by-1 vector of monotonically increasing values

Specify the breakpoint data explicitly or as evenly-spaced breakpoints, based on the value of the Breakpoints specification parameter:

- If you set Breakpoints specification to Explicit values, enter the breakpoint set that corresponds to each dimension of table data in each Breakpoints row. For each dimension, specify breakpoints as a 1-by-n or n-by-1 vector whose values are strictly monotonically increasing.
- If you set Breakpoints specification to Even spacing, enter the parameters First point and Spacing in each Breakpoints row to generate evenly-spaced breakpoints in the respective dimension. Your table data determines the number of evenly spaced points.

**Dependencies**

To enable this parameter, set Data specification to Table and breakpoints.

**Programmatic Use**

**Block Parameter:** BreakpointsForDimension1
**Type:** character vector  
**Values:** 1-by-n or n-by-1 vector of monotonically increasing values  
**Default:** ' [1:3] '

**First point — First point in evenly spaced breakpoint data**  
1 (default) | scalar

Specify the first point in your evenly spaced breakpoint data as a real-valued, finite, scalar. This parameter is available when **Breakpoints specification** is set to **Even spacing**.

**Dependencies**

To enable this parameter, set **Data specification** to Table and breakpoints, and **Breakpoints specification** to Even spacing.

**Programmatic Use**  
**Block Parameter:** BreakpointsForDimension1FirstPoint | BreakpointsForDimension2FirstPoint  
**Type:** character vector  
**Values:** real-valued, finite, scalar  
**Default:** ' 1 '

**Spacing — Spacing between evenly spaced breakpoints**  
1 (default) | scalar

Specify the spacing between points in your evenly spaced breakpoint data.

**Dependencies**

To enable this parameter, set **Data specification** to Table and breakpoints, and **Breakpoints specification** to Even spacing.

**Programmatic Use**  
**Block Parameter:** BreakpointsForDimension1Spacing | BreakpointsForDimension2Spacing  
**Type:** character vector  
**Values:** positive, real-valued, finite, scalar  
**Default:** ' 1 '

**Edit table and breakpoints — Launch Lookup Table Editor dialog box**

button
Click this button to open the Lookup Table Editor. For more information, see “Edit Lookup Tables” in the Simulink documentation.

Clicking this button for a lookup table object lets you edit the object and save the new values for the object.

**Algorithm**

**Lookup method**

Interpolation method — Method of interpolation between breakpoint values

Linear point-slope (default) | Flat | Nearest | Linear Lagrange | Cubic spline | Akima spline

When an input falls between breakpoint values, the block interpolates the output value using neighboring breakpoints. For more information on interpolation methods, see “Interpolation Methods”.

**Dependencies**

- If you select Cubic spline, the block supports only scalar signals. The other interpolation methods support nonscalar signals.
- If you select Akima spline, the extrapolation method can only be Akima spline.

When set to the modified Akima interpolation method, this block does not support:

- Row-major array layout and algorithms optimized for row-major array layout
- Scaled double and fixed-point data types
- Simulink.LookupTable objects
- Code generation when the configuration parameter Code Generation > Interface > Support non-finite numbers check box is selected

**Programmatic Use**

**Block Parameter:** InterpMethod

**Type:** character vector

**Values:** 'Linear point-slope' | 'Flat' | 'Nearest' | 'Linear Lagrange' | 'Cubic spline' | 'Akima spline'

**Default:** 'Linear point-slope'
Extrapolation method — Method of handling input values that fall outside the range of a breakpoint data set
Clip (default) | Linear | Cubic spline | Akima spline

Select Clip, Linear, or Cubic spline. See “Extrapolation Methods” for more information.

If the extrapolation method is Linear, the extrapolation value is calculated based on the selected linear interpolation method. For example, if the interpolation method is linear Lagrange, the extrapolation method inherits the linear Lagrange equation to compute the extrapolated value.

Dependencies

• To select Cubic spline for Extrapolation method, you must also select Cubic spline for Interpolation method.
• To select Akima spline for Extrapolation method, you must also select Akima spline for Interpolation method.

Programmatic Use

Block Parameter: ExtrapMethod
Type: character vector
Values: 'Linear' | 'Clip' | 'Cubic spline' | 'Akima spline'
Default: 'Linear'

Index search method — Method of calculating table indices
Evenly spaced points (default) | Linear search | Binary search

Select Evenly spaced points, Linear search, or Binary search. Each search method has speed advantages in different circumstances:

• For evenly spaced breakpoint sets (for example, 10, 20, 30, and so on), you achieve optimal speed by selecting Evenly spaced points to calculate table indices.

This algorithm uses only the first two breakpoints of a set to determine the offset and spacing of the remaining points.

Note Set Index search method to Evenly spaced points when using the Simulink.LookupTable object to specify table data and the Breakpoints Specification parameter of the referenced Simulink.LookupTable object is set to Even spacing.
• For unevenly spaced breakpoint sets, follow these guidelines:
  • If input signals do not vary much between time steps, selecting Linear search with Begin index search using previous index result produces the best performance.
  • If input signals jump more than one or two table intervals per time step, selecting Binary search produces the best performance.

A suboptimal choice of index search method can lead to slow performance of models that rely heavily on lookup tables.

**Note** The generated code stores only the first breakpoint, the spacing, and the number of breakpoints when:
• The breakpoint data is not tunable.
• The index search method is Evenly spaced points.

---

**Programmatic Use**

**Block Parameter:** IndexSearchMethod  
**Type:** character vector  
**Values:** 'Binary search' | 'Evenly spaced points' | 'Linear search'  
**Default:** 'Binary search'

**Begin index search using previous index result — Start using the index from the previous time step**  
off (default) | on

Select this check box when you want the block to start its search using the index found at the previous time step. For inputs that change slowly with respect to the interval size, enabling this option can improve performance. Otherwise, the linear search and binary search methods can take longer, especially for large breakpoint sets.

**Dependencies**

To enable this parameter, set Index search method to Linear search or Binary search.

**Programmatic Use**

**Block Parameter:** BeginIndexSearchUsingPreviousIndexResult
**Type:** character vector  
**Values:** 'off' | 'on'  
**Default:** 'off'

Diagnostic for out-of-range input — Block action when input is out of range

Specify whether to produce a warning or error when the input is out of range. Options include:

- **None** — Produce no response.
- **Warning** — Display a warning and continue the simulation.
- **Error** — Terminate the simulation and display an error.

Programmatic Use

**Block Parameter:** DiagnosticForOutOfRangeInput  
**Type:** character vector  
**Values:** 'None' | 'Warning' | 'Error'  
**Default:** 'None'

Use last table value for inputs at or above last breakpoint — Method for computing output for inputs at or above last breakpoint

Using this check box, specify the indexing convention that the block uses to address the last element of a breakpoint set and its corresponding table value. This check box is relevant if the input is equal to or larger than the last element of the breakpoint data. Due to rounding, selecting and clearing this check box may result in differing results for the last breakpoint between simulation and code generation.

<table>
<thead>
<tr>
<th>Check Box</th>
<th>Index Used by Block</th>
<th>Interval Fraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selected</td>
<td>Last element of breakpoint data on the Table and Breakpoints tab</td>
<td>0</td>
</tr>
<tr>
<td>Cleared</td>
<td>Next-to-last element of breakpoint data on the Table and Breakpoints tab</td>
<td>1</td>
</tr>
</tbody>
</table>

Given an input $u$ within range of a breakpoint set $bp$, the interval fraction $f$, in the range $0 < f < 1$, is computed as shown below.
Suppose the breakpoint set is [1 4 5] and input \( u \) is 5.5. If you select this check box, the index is that of the last element (5) and the interval fraction is 0. If you clear this check box, the index is that of the next-to-last element (4) and the interval fraction is 1.

**Dependencies**

To enable this parameter, set:

- **Interpolation method** to **Linear**.
- **Extrapolation method** to **Clip**.

**Programmatic Use**

**Block Parameter:** `UseLastTableValue`

- **Type:** character vector
- **Values:** 'off' | 'on'
- **Default:** 'off'

**Input settings**

**Use one input port for all input data — Use only one input port**

- **off** (default) | **on**

Select this check box to use only one input port that expects a signal that is \( n \) elements wide for an \( n \)-dimensional table. This option is useful for removing line clutter on a block diagram with many lookup tables.
Note  When you select this check box, one input port with the label u appears on the block.

Programmatic Use
Block Parameter: UseOneInputPortForAllInputData
Type: character vector
Values: 'off' | 'on'
Default: 'off'

Code generation

Remove protection against out-of-range input in generated code —
Remove code that checks for out-of-range input values
off (default) | on

Specify whether or not to include code that checks for out-of-range input values.

<table>
<thead>
<tr>
<th>Check Box</th>
<th>Result</th>
<th>When to Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>on</td>
<td>Generated code does not include conditional statements to check for out-of-range breakpoint inputs.</td>
<td>For code efficiency</td>
</tr>
<tr>
<td></td>
<td>When the input is out-of-range, it may cause undefined behavior for generated code and simulations using accelerator mode.</td>
<td></td>
</tr>
<tr>
<td>off</td>
<td>Generated code includes conditional statements to check for out-of-range inputs.</td>
<td>For safety-critical applications</td>
</tr>
</tbody>
</table>

If your input is not out of range, you can select the Remove protection against out-of-range index in generated code check box for code efficiency. By default, this check box is cleared. For safety-critical applications, do not select this check box. If you want to select the Remove protection against out-of-range index in generated code check box, first check that your model inputs are in range. For example:
1. Clear the **Remove protection against out-of-range index in generated code** check box.

2. Set the **Diagnostic for out-of-range input** parameter to **Error**.

3. Simulate the model in normal mode.

4. If there are out-of-range errors, fix them to be in range and run the simulation again.

5. When the simulation no longer generates out-of-range input errors, select the **Remove protection against out-of-range index in generated code** check box.

**Note** When you select the **Remove protection against out-of-range index in generated code** check box and the input \( k \) or \( f \) is out of range, the behavior is undefined for generated code and simulations using accelerator mode.

Depending on your application, you can run the following Model Advisor checks to verify the usage of this check box:

- **By Product > Embedded Coder > Identify lookup table blocks that generate expensive out-of-range checking code**
- **By Product > Simulink Check > Modeling Standards > DO-178C/DO-331 Checks > Check usage of lookup table blocks**

For more information about the Model Advisor, see “Run Model Advisor Checks”.

**Programmatic Use**

**Block Parameter:** RemoveProtectionInput  
**Type:** character vector  
**Values:** 'off' | 'on'  
**Default:** 'off'

**Support tunable table size in code generation — Enable tunable table size in the generated code**

off (default) | on

Select this check box to enable tunable table size in the generated code. This option enables you to change the size and values of the lookup table and breakpoint data in the generated code without regenerating or recompiling the code. You can only decrease the size of the lookup table and breakpoint data.

**Dependencies**

If you set **Interpolation method** to Cubic spline, this check box is not available.
Programmatic Use
Block Parameter: SupportTunableTableSize
Type: character vector
Values: 'off' | 'on'
Default: 'off'

Sample time — Specify sample time as a value other than -1
-1 (default) | scalar | vector

Specify the sample time as a value other than -1. For more information, see “Specify Sample Time”.

Dependencies
This parameter is not visible unless it is explicitly set to a value other than -1. To learn more, see “Blocks for Which Sample Time Is Not Recommended”.

Programmatic Use
Block Parameter: SampleTime
Type: character vector
Values: scalar or vector
Default: '-1'

Maximum indices for each dimension — Maximum index value for each table dimension
[] (default) | scalar or vector of positive integer values

Example: [4 6] for a 5-by-7 table

Specify the maximum index values for each table dimension using zero-based indexing. You can specify a scalar or vector of positive integer values using the following data types:

- Built-in floating-point types: double and single
- Built-in integer types: int8, int16, int32, uint8, uint16, and uint32

Examples of valid specifications include:

- [4 6] for a 5-by-7 table
- [int8(2) int16(5) int32(9)] for a 3-by-6-by-10 table
- A Simulink.Parameter whose value on generating code is one less than the dimensions of the table data. For more information, see “Tunable Table Size in the Generated Code” on page 1-1172.
Dependencies

To enable this parameter, select **Support tunable table size in code generation**. On tuning this parameter in the generated code, provide the new table data and breakpoints along with the tuned parameter value.

Programmatic Use

**Block Parameter:** MaximumIndicesForEachDimension  
**Type:** character vector  
**Values:** scalar or vector of positive integer values  
**Default:** '[]'

Data Types

**Table data — Data type of table data**

Inherit: Same as output (default) | double | single | int8 | uint8 | int16 | uint16 | int32 | uint32 | int64 | uint64 | fixdt(1,16) | fixdt(1,16,0) | fixdt(1,16,2^0,0) | <data type expression>

Specify the table data type. You can set it to:

- A rule that inherits a data type, for example, Inherit: Same as output
- The name of a built-in data type, for example, single
- The name of a data type object, for example, a Simulink.NumericType object
- An expression that evaluates to a data type, for example, fixdt(1,16,0)

Click the **Show data type assistant** button to display the **Data Type Assistant**, which helps you set the data type attributes. For more information, see “Specify Data Types Using Data Type Assistant”.

**Tip** Specify a table data type different from the output data type for these cases:

- Lower memory requirement for storing table data that uses a smaller type than the output signal
- Sharing of prescaled table data between two n-D Lookup Table blocks with different output data types
- Sharing of custom storage table data in the generated code for blocks with different output data types
Programmatic Use
Block Parameter: TableDataTypeStr
Type: character vector
Values: 'Inherit: Inherit from 'Table data'' | 'Inherit: Same as output' | 'double' | 'single' | 'int8' | 'uint8' | 'int16' | 'uint16' | 'int32' | 'uint32' | 'int64' | 'uint64' | 'fixdt(1,16)' | 'fixdt(1,16,0)' | 'fixdt(1,16,2^0,0)' | '<data type expression>'
Default: 'Inherit: Same as output'

Table data Minimum — Minimum value of the table data
[] | scalar
Specify the minimum value for table data. The default value is [] (unspecified).

Programmatic Use
Block Parameter: TableMin
Type: character vector
Values: scalar
Default: '

Table data Maximum — Maximum value of the table data
[] | scalar
Specify the maximum value for table data. The default value is [] (unspecified).

Programmatic Use
Block Parameter: TableMax
Type: character vector
Values: scalar
Default: '

Breakpoints — Breakpoint data type
Inherit: Same as corresponding input (default) | double | single | int8 | uint8 | int16 | uint16 | int32 | uint32 | int64 | uint64 | fixdt(1,16) | fixdt(1,16,0) | fixdt(1,16,2^0,0) | Enum: <class name> | <data type expression>
Specify the data type for a set of breakpoint data. You can set it to:

- A rule that inherits a data type, for example, Inherit: Same as corresponding input
- The name of a built-in data type, for example, single
• The name of a data type class, for example, an enumerated data type class
• The name of a data type object, for example, a Simulink.NumericType object
• An expression that evaluates to a data type, for example, fixdt(1,16,0)

**Tip**

• Breakpoints support unordered enumerated data. As a result, linear searches are also unordered, which offers flexibility but can impact performance. The search begins from the first element in the breakpoint.
• If the **Begin index search using previous index result** check box is selected, you must use ordered monotonically increasing data. This ordering improves performance.
• For enumerated data, **Extrapolation method** must be Clip.
• The block does not support out-of-range input for enumerated data. When specifying enumerated data, include the entire enumeration set in the breakpoint data set. For example, use the enumeration function.

This is a limitation for using enumerated data with this block:

• The block does not support out-of-range input for enumerated data. When specifying enumerated data, include the entire enumeration set in the breakpoint data set. For example, use the enumeration function.

Click the **Show data type assistant** button to display the **Data Type Assistant**, which helps you set the data type attributes. For more information, see “Specify Data Types Using Data Type Assistant”.

**Tip** Specify a breakpoint data type different from the corresponding input data type for these cases:

• Lower memory requirement for storing breakpoint data that uses a smaller type than the input signal
• Sharing of prescaled breakpoint data between two n-D Lookup Table blocks with different input data types
• Sharing of custom storage breakpoint data in the generated code for blocks with different input data types
Programmatic Use
Block Parameter: BreakpointsForDimension1DataTypeStr | BreakpointsForDimension2DataTypeStr| ... | BreakpointsForDimension30DataTypeStr
Type: character vector
Values: 'Inherit: Same as corresponding input' | 'Inherit: Inherit from 'Breakpoint data'' | 'double' | 'single' | 'int8' | 'uint8' | 'int16' | 'uint16' | 'int32' | 'uint32' | 'int64' | 'uint64' | 'fixdt(1,16)' | 'fixdt(1,16,0)' | 'fixdt(1,16,2^0,0)''<data type expression>'
Default: 'Inherit: Same as corresponding input'

Breakpoints Minimum — Minimum value breakpoint data can have
[] | scalar
Specify the minimum value that a set of breakpoint data can have. The default value is [] (unspecified).

Programmatic Use
Block Parameter: BreakpointsForDimension1Min | BreakpointsForDimension2Min | ... | BreakpointsForDimension30Min
Type: character vector
Values: scalar
Default: '

Breakpoints Maximum — Maximum value breakpoint data can have
[] | scalar
Specify the maximum value that a set of breakpoint data can have. The default value is [] (unspecified).

Programmatic Use
Block Parameter: BreakpointsForDimension1Max | BreakpointsForDimension2Max | ... | BreakpointsForDimension30Max
Type: character vector
Values: scalar
Default: '

Fraction — Fraction data type
Inherit: Inherit via internal rule (default) | double | single | fixdt(1,16,0)''<data type expression>'
Specify the fraction data type. You can set it to:
• A rule that inherits a data type, for example, **Inherit: Inherit via internal rule**
• The name of a built-in data type, for example, **single**
• The name of a data type object, for example, a Simulink.NumericType object
• An expression that evaluates to a data type, for example, **fixdt(1,16,0)**

Click the **Show data type assistant** button to display the **Data Type Assistant**, which helps you set the data type attributes. For more information, see “Specify Data Types Using Data Type Assistant”.

**Programmatic Use**

**Block Parameter:** FractionDataTypeStr  
**Type:** character vector  
**Values:** 'Inherit: Inherit via internal rule' | 'double' | 'single' | 'fixdt(1,16,0)' | '<data type expression>'  
**Default:** 'Inherit: Inherit via internal rule'

**Intermediate results — Intermediate results data type**

**Inherit:** Same as output (default) | double | single | int8 | uint8 | int16 | uint16 | int32 | uint32 | int64 | uint64 | fixdt(1,16) | fixdt(1,16,0) | fixdt(1,16,2^0,0) | <data type expression>

Specify the intermediate results data type. You can set it to:

• A rule that inherits a data type, for example, **Inherit: Same as output**
• The name of a built-in data type, for example, **single**
• The name of a data type object, for example, a Simulink.NumericType object
• An expression that evaluates to a data type, for example, **fixdt(1,16,0)**

Click the **Show data type assistant** button to display the **Data Type Assistant**, which helps you set the data type attributes. For more information, see “Specify Data Types Using Data Type Assistant”.

**Tip** Use this parameter to specify higher (or lower) precision for internal computations than for table data or output data.
**Programmatic Use**

**Block Parameter:** IntermediateResultsDataTypeStr  
**Type:** character vector  
**Values:**  
- 'Inherit: Inherit via internal rule'  
- 'Inherit: Same as output'  
- 'double'  
- 'single'  
- 'int8'  
- 'uint8'  
- 'int16'  
- 'uint16'  
- 'int32'  
- 'uint32'  
- 'int64'  
- 'uint64'  
- 'fixdt(1,16,0)'  
- 'fixdt(1,16,2^0,0)'  
- '<data type expression>'  
**Default:** 'Inherit: Same as output'

**Output — Output data type**

Inherit: Same as input (default) | double | single | int8 | uint8 | int16 | uint16 | int32 | uint32 | int64 | uint64 | fixdt(1,16) | fixdt(1,16,0) | fixdt(1,16,2^0,0) | '<data type expression>'

Specify the output data type. You can set it to:

- A rule that inherits a data type, for example, Inherit: Inherit via back propagation
- The name of a built-in data type, for example, single
- The name of a data type object, for example, a Simulink.NumericType object
- An expression that evaluates to a data type, for example, fixdt(1,16,0)

Click the **Show data type assistant** button to display the **Data Type Assistant**, which helps you set the data type attributes. For more information, see “Specify Data Types Using Data Type Assistant”.

**Programmatic Use**

**Block Parameter:** OutDataTypeStr  
**Type:** character vector  
**Values:**  
- 'Inherit: Inherit via back propagation'  
- 'Inherit: Inherit from table data'  
- 'Inherit: Same as first input'  
- 'double'  
- 'single'  
- 'int8'  
- 'uint8'  
- 'int16'  
- 'uint16'  
- 'int32'  
- 'uint32'  
- 'int64'  
- 'uint64'  
- 'fixdt(1,16,0)'  
- 'fixdt(1,16,2^0,0)'  
- '<data type expression>'  
**Default:** 'Inherit: Same as first input'

**Output Minimum — Minimum value the block can output**

[] | scalar
Specify the minimum value that the block outputs. The default value is [ ] (unspecified). Simulink software uses this value to perform:

- Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters”).
- Simulation range checking (see “Signal Ranges”).
- Automatic scaling of fixed-point data types.
- Optimization of the code that you generate from the model. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. For more information, see “Optimize using the specified minimum and maximum values” (Simulink Coder).

**Programmatic Use**

**Block Parameter:** OutMin  
**Type:** character vector  
**Values:** scalar  
**Default:** ' [ ] '

**Output Maximum — Maximum value the block can output**  
[[ ] | scalar]

Specify the maximum value that the block can output. The default value is [ ] (unspecified). Simulink software uses this value to perform:

- Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters”).
- Simulation range checking (see “Signal Ranges”).
- Automatic scaling of fixed-point data types.
- Optimization of the code that you generate from the model. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. For more information, see “Optimize using the specified minimum and maximum values” (Simulink Coder).

**Programmatic Use**

**Block Parameter:** OutMax  
**Type:** character vector  
**Values:** scalar  
**Default:** ' [ ] '
**Internal rule priority — Internal rule for intermediate calculations**

Specify the internal rule for intermediate calculations. Select Speed for faster calculations. If you do, a loss of accuracy might occur, usually up to 2 bits.

**Programmatic Use**

**Block Parameter:** InternalRulePriority  
**Type:** character vector  
**Values:** 'Speed' | 'Precision'  
**Default:** 'Speed'

**Require all inputs to have the same data type — Require all inputs to have the same data type**

Select to require all inputs to have the same data type.

**Programmatic Use**

**Block Parameter:** InputSameDT  
**Type:** character vector  
**Values:** 'off' | 'on'  
**Default:** 'on'

**Lock data type settings against changes by the fixed-point tools — Prevent fixed-point tools from overriding data types**

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on this block. For more information, see “Lock the Output Data Type Setting” (Fixed-Point Designer).

**Programmatic Use**

**Block Parameter:** LockScale  
**Type:** character vector  
**Values:** 'off' | 'on'  
**Default:** 'off'

**Integer rounding mode — Rounding mode for fixed-point operations**

Simplest (default) | Ceiling | Convergent | Floor | Nearest | Round | Zero
Specify the rounding mode for fixed-point lookup table calculations that occur during simulation or execution of code generated from the model. For more information, see “Rounding” (Fixed-Point Designer).

This option does not affect rounding of values of block parameters. Simulink rounds such values to the nearest representable integer value. To control the rounding of a block parameter, enter an expression using a MATLAB rounding function into the edit field on the block dialog box.

**Programmatic Use**

**Block Parameter:** RndMeth

**Type:** character vector

**Values:** 'Ceiling' | 'Convergent' | 'Floor' | 'Nearest' | 'Round' | 'Simplest' | 'Zero'

**Default:** 'Simplest'

---

**Saturate on integer overflow — Method of overflow action**

**off (default) | on**

<table>
<thead>
<tr>
<th>Action</th>
<th>Reasons for Taking This Action</th>
<th>What Happens for Overflows</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Select this check box (on).</td>
<td>Your model has possible overflow and you want explicit saturation protection in the generated code.</td>
<td>Overflows saturate to either the minimum or maximum value that the data type can represent.</td>
<td>An overflow associated with a signed 8-bit integer can saturate to -128 or 127.</td>
</tr>
<tr>
<td>Do not select this check box (off).</td>
<td>You want to optimize efficiency of your generated code.</td>
<td>Overflows wrap to the appropriate value that is representable by the data type.</td>
<td>The number 130 does not fit in a signed 8-bit integer and wraps to -126.</td>
</tr>
</tbody>
</table>

Tip  If you save your model as version R2009a or earlier, this check box setting has no effect and no saturation code appears. This behavior preserves backward compatibility.
When you select this check box, saturation applies to every internal operation on the block, not just the output or result. In general, the code generation process can detect when overflow is not possible. In this case, the code generator does not produce saturation code.

**Programmatic Use**

**Block Parameter:** SaturateOnIntegerOverflow  
**Type:** character vector  
**Values:** 'off' | 'on'  
**Default:** 'off'

### Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>double</th>
<th>enumerated</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### More About

#### Tunable Table Size in the Generated Code

Suppose that you have a lookup table and want to make the size tunable in the generated code. When you use `Simulink.LookupTable` and `Simulink.Breakpoint` objects to configure lookup table data for calibration in the generated code, use the `SupportTunableSize` property of the objects to enable a tunable table size. When you do not use these classes, use the **Support tunable table size in code generation** parameter in an n-D Lookup Table block to enable a tunable table size.

Assume that:
You define a `Simulink.Parameter` structure in the preload function of your model:

```matlab
p = Simulink.Parameter;
p.Value.MaxIdx = [2 2];
p.Value.BP1 = [1 2 3];
p.Value.BP2 = [1 4 16];
p.Value.Table = [4 5 6; 16 19 20; 10 18 23];
p.DataType = 'Bus: slLookupTable';
p.CoderInfo.StorageClass = 'ExportedGlobal';
```

% Create bus object slBus1 from MATLAB structure
Simulink.Bus.createObject(p.Value);
slLookupTable = slBus1;
slLookupTable.Elements(1).DataType = 'uint32';

These block parameters apply in the n-D Lookup Table block.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of table dimensions</td>
<td>2</td>
</tr>
<tr>
<td>Table data</td>
<td>p.Table</td>
</tr>
<tr>
<td>Breakpoints 1</td>
<td>p.BP1</td>
</tr>
<tr>
<td>Breakpoints 2</td>
<td>p.BP2</td>
</tr>
<tr>
<td>Support tunable table size in code generation</td>
<td>on</td>
</tr>
<tr>
<td>Maximum indices for each dimension</td>
<td>p.MaxIdx</td>
</tr>
</tbody>
</table>

The generated `model_types.h` header file contains a type definition that looks something like this.

```c
typedef struct {
    uint32_T MaxIdx[2];
    real_T BP1[3];
    real_T BP2[3];
    real_T Table[9];
} slLookupTable;
```

The generated `model.c` file contains code that looks something like this.

```c
/* Exported block parameters */
slLookupTable p = {
    { 2U, 2U },
```
{ 1.0, 2.0, 3.0 },
{ 1.0, 4.0, 16.0 },
{ 4.0, 16.0, 10.0, 5.0, 19.0, 18.0, 6.0, 20.0, 23.0 }
};

/* More code */

/* Model output function */
static void ex_lut_nd_tunable_table_output(int_T tid)
{
    /* Lookup n-D: '<Root>/n-D Lookup Table' incorporates:
    * Inport: '<Root>/In1'
    * Inport: '<Root>/In2'
    */
    Y = look2_binlcpw(U1, U2, p.BP1, p.BP2, p.Table, ...
                        p.MaxIdx, p.MaxIdx[0] + 1U);

    /* Outport: '<Root>/Out1' */
    ex_lut_nd_tunable_table_Y.Out1 = Y;

    /* tid is required for a uniform function interface.
    * Argument tid is not used in the function. */
    UNUSED_PARAMETER(tid);
}

The highlighted line of code specifies a tunable table size for the lookup table. You can change the size and values of the lookup table and breakpoint data without regenerating or recompiling the code.

**Enumerated Values in Lookup Tables**

Suppose that you have a lookup table with an enumerated class like this defined:

classdef(Enumeration) Gears < Simulink.IntEnumType
    enumeration
        GEAR1(1),
        GEAR2(2),
        GEAR3(4),
        GEAR4(8),
        SPORTS(16),
        REVERSE(-1),
n-D Lookup block has these settings:

- **Number of dimensions** to 1.
- **Table data** value is \([5 \ 10 \ 20 \ 40 \ 80 \ -5 \ 0]\).
- **Breakpoints 1** value is enumeration('Gears').
- Interpolation method is Flat.
- For an unordered search, set **Index search method** to **Linear search** and clear the **Begin index search using previous index result** check box.

Simulation produces a vector \([10 \ -5 \ 80]\), which correspond to GEAR2, REVERSE, and SPORTS.

**Extended Capabilities**

**C/C++ Code Generation**
Generate C and C++ code using Simulink® Coder™.

**HDL Code Generation**
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic. For information about HDL code generation support for 2-D Lookup Table blocks, see “HDL Code Generation” on page 1-1205.
Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

See Also
Direct Lookup Table (n-D) | Interpolation Using Prelookup | Lookup Table Dynamic | Prelookup | Simulink.Breakpoint | Simulink.LookupTable

Topics
“Import Lookup Table Data from MATLAB”
“About Lookup Table Blocks”
“Anatomy of a Lookup Table”
“Enter Breakpoints and Table Data”
“Guidelines for Choosing a Lookup Table”

Introduced in R2011a
n-D Lookup Table

Approximate n-dimensional function

Library:
- Simulink / Lookup Tables
- HDL Coder / Lookup Tables

Description

Supported Block Operations

The 1-D, 2-D, and n-D Lookup Table blocks evaluate a sampled representation of a function in N variables

\[ y = F(x_1, x_2, x_3, ..., x_N) \]

where the function \( F \) can be empirical. The block maps inputs to an output value by looking up or interpolating a table of values you define with block parameters. The block supports flat (constant), linear (linear point-slope), Lagrange (linear Lagrange), nearest, cubic-spline, and Akima spline interpolation methods. You can apply these methods to a table of any dimension from 1 through 30.

In the following block, the first input identifies the first dimension (row) breakpoints, the second input identifies the second dimension (column) breakpoints, and so on.

See “Port Location After Rotating or Flipping” for a description of the port order for various block orientations.
When the **Math and Data Types > Use algorithms optimized for row-major array layout** configuration parameter is set, the 2-D and n-D Lookup Table block behavior changes from column-major to row-major. For these blocks, the column-major and row-major algorithms may differ in the order of the output calculations, possibly resulting in slightly different numerical values. This capability requires a Simulink Coder or Embedded Coder license. For more information on row-major support, see “Code Generation of Matrices and Arrays” (Simulink Coder).

### Specification of Breakpoint and Table Data

These block parameters define the breakpoint and table data.

<table>
<thead>
<tr>
<th>Block Parameter</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Number of table dimensions</strong></td>
<td>Specifies the number of dimensions of your lookup table.</td>
</tr>
<tr>
<td><strong>Breakpoints</strong></td>
<td>Specifies a breakpoint vector that corresponds to each dimension of your lookup table.</td>
</tr>
<tr>
<td><strong>Table data</strong></td>
<td>Defines the associated set of output values.</td>
</tr>
</tbody>
</table>

**Tip** Evenly spaced breakpoints can make the generated code division-free. For more information, see `fixpt_evenspace_cleanup` and “Identify questionable fixed-point operations” (Embedded Coder).

### How the Block Generates Output

The n-D, 1-D and 2-D Lookup Table blocks generate output by looking up or estimating table values based on the input values.

<table>
<thead>
<tr>
<th>Block Inputs</th>
<th>n-D Lookup Table Block Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>Match the values of indices in breakpoint data sets</td>
<td>Outputs the table value at the intersection of the row, column, and higher dimension breakpoints</td>
</tr>
<tr>
<td>Do not match the values of indices in breakpoint data sets, but are within range</td>
<td>Interpolates appropriate table values, using the <strong>Interpolation method</strong> you select</td>
</tr>
</tbody>
</table>
**Block Inputs**

<table>
<thead>
<tr>
<th>n-D Lookup Table Block Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>Do not match the values of indices in breakpoint data sets, and are out of range</td>
</tr>
</tbody>
</table>

**Other Blocks that Perform Equivalent Operations**

You can use the Interpolation Using Prelookup block with the Prelookup block to perform the equivalent operation of one n-D Lookup Table block. This combination of blocks offers greater flexibility that can result in more efficient simulation performance for linear interpolations.

When the lookup operation is an array access that does not require interpolation, use the Direct Lookup Table (n-D) block. For example, if you have an integer value k and you want the kth element of a table, \( y = \text{table}(k) \), interpolation is unnecessary.

**Ports**

**Input**

**u1 — First-dimension (row) inputs**

- scalar | vector | matrix

Real-valued inputs to the u1 port, mapped to an output value by looking up or interpolating the table of values that you define.

Example: 0:10

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | enumerated | fixed point

**uN — n-th dimension input values**

- scalar | vector | matrix

Real-valued inputs to the uN port, mapped to an output value by looking up or interpolating the table of values that you define.

Example: 0:10

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | enumerated | fixed point
**Output**

Port_1 — Output computed by looking up or estimating table values

scalar | vector | matrix

Output generated by looking up or estimating table values based on the input values:

<table>
<thead>
<tr>
<th>When block inputs...</th>
<th>The n-D Lookup Table block...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Match the values of indices in breakpoint data sets</td>
<td>Outputs the table value at the intersection of the row, column, and higher dimension breakpoints</td>
</tr>
<tr>
<td>Do not match the values of indices in breakpoint data sets, but are within range</td>
<td>Interpolates appropriate table values, using the <strong>Interpolation method</strong> you select</td>
</tr>
<tr>
<td>Do not match the values of indices in breakpoint data sets, and are out of range</td>
<td>Extrapolates the output value, using the <strong>Extrapolation method</strong> you select</td>
</tr>
</tbody>
</table>

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

**Parameters**

**Table and Breakpoints**

**Number of table dimensions — Number of lookup table dimensions**

3 (default) | 1 | 2 | 4 | ... | 30

Enter the number of dimensions of the lookup table. This parameter determines:

- The number of independent variables for the table and the number of block inputs
- The number of breakpoint sets to specify

<table>
<thead>
<tr>
<th>Value</th>
<th>Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>1, 2, 3, or 4</td>
<td>Select the value from the drop-down list.</td>
</tr>
</tbody>
</table>
Value | Setting
---|---
A higher number of table dimensions | Enter a positive integer directly in the field. The maximum number of table dimensions that this block supports is 30.

For example, a table with a size of $M \times N \times \ldots$ means that the size of dimension 1 is $M$, the size of dimension 2 is $N$, and so forth. $M$ must match the first breakpoint length, $N$ must match the second breakpoint length, and so forth.

**Programmatic Use**

**Block Parameter:** NumberOfTableDimensions  
**Type:** character vector  
**Values:** '1' | '2' | '3' | '4' | ... | 30  
**Default:** '3'

**Data specification — Method of table and breakpoint specification**  
Table and breakpoints (default) | Lookup table object

From the list, select:

- **Table and breakpoints** — Specify the table data and breakpoints. Selecting this option enables the following parameters:
  - **Table data**
  - **Breakpoints specification**
  - **Breakpoints 1**
  - **Breakpoints 2**
  - **Breakpoints 3**
  - **Edit table and breakpoints**
- **Lookup table object** — Use an existing lookup table (`Simulink.LookupTable`) object. Selecting this option enables the **Name** field and **Edit table and breakpoints** button.

**Programmatic Use**

**Block Parameter:** DataSpecification  
**Type:** character vector  
**Values:** 'Table and breakpoints' | 'Lookup table object'  
**Default:** 'Table and breakpoints'
Name — Name of the lookup table object

Enter the name of the lookup table (Simulink.LookupTable) object. If a 
Simulink.LookupTable object does not exist, click the action button and select Create. The corresponding parameters of the new lookup table object are automatically populated with the block information.

Dependencies

To enable this parameter, set Data specification to Lookup table object.

Programmatic Use

Block Parameter: LookupTableObject
Type: character vector
Values: name of a Simulink.LookupTable object
Default: ''

Table data — Define the table of output values

Enter the table of output values.

During simulation, the matrix size must match the dimensions defined by the Number of table dimensions parameter. However, during block diagram editing, you can enter an empty matrix (specified as []) or an undefined workspace variable. This technique lets you postpone specifying a correctly dimensioned matrix for the table data and continue editing the block diagram.

Dependencies

To enable this parameter, set Data specification to Table and breakpoints.

Programmatic Use

Block Parameter: Table
Type: character vector
Values: matrix of table values
Default: 'reshape(repmat([4 5 6;16 19 20;10 18 23],1,2),[3,3,2])'

Breakpoints specification — Method of breakpoint specification

Explicit values (default) | Even spacing
Specify whether to enter data as explicit breakpoints or as parameters that generate evenly spaced breakpoints.

- To explicitly specify breakpoint data, set this parameter to Explicit values and enter breakpoint data in the text box next to the Breakpoints parameters.
- To specify parameters that generate evenly spaced breakpoints, set this parameter to Even spacing and enter values for the First point and Spacing parameters for each dimension of breakpoint data. The block calculates the number of points to generate from the table data.

Dependencies

To enable this parameter, set Data specification to Table and breakpoints.

Programmatic Use

Block Parameter: BreakpointsSpecification
Type: character vector
Values: 'Explicit values' | 'Even spacing'
Default: 'Explicit values'

Breakpoints — Explicit breakpoint values, or first point and spacing of breakpoints
[10,22,31] (default) | 1-by-n or n-by-1 vector of monotonically increasing values

Specify the breakpoint data explicitly or as evenly-spaced breakpoints, based on the value of the Breakpoints specification parameter.

- If you set Breakpoints specification to Explicit values, enter the breakpoint set that corresponds to each dimension of table data in each Breakpoints row. For each dimension, specify breakpoints as a 1-by-n or n-by-1 vector whose values are strictly monotonically increasing.
- If you set Breakpoints specification to Even spacing, enter the parameters First point and Spacing in each Breakpoints row to generate evenly-spaced breakpoints in the respective dimension. Your table data determines the number of evenly spaced points.

Dependencies

To enable this parameter, set Data specification to Table and breakpoints.
Programmatic Use
Block Parameter: BreakpointsForDimension1 | BreakpointsForDimension2 | ... | BreakpointsForDimension30 | Type: character vector
Values: 1-by-n or n-by-1 vector of monotonically increasing values
Default: '[10, 22, 31]'

First point — First point in evenly spaced breakpoint data
1 (default) | scalar

Specify the first point in your evenly spaced breakpoint data as a real-valued, finite scalar. This parameter is available when Breakpoints specification is set to Even spacing.

Dependencies
To enable this parameter, set Data specification to Table and breakpoints, and Breakpoints specification to Even spacing.

Programmatic Use
Block Parameter: BreakpointsForDimension1FirstPoint | BreakpointsForDimension2FirstPoint | ... | BreakpointsForDimension30FirstPoint | Type: character vector
Values: real-valued, finite, scalar
Default: '1'

Spacing — Spacing between evenly spaced breakpoints
1 (default) | scalar

Specify the spacing between points in your evenly-spaced breakpoint data.

Dependencies
To enable this parameter, set Data specification to Table and breakpoints, and Breakpoints specification to Even spacing.

Programmatic Use
Block Parameter: BreakpointsForDimension1Spacing | BreakpointsForDimension2Spacing | ... | BreakpointsForDimension30Spacing | Type: character vector
Values: positive, real-valued, finite, scalar
Default: '1'
**Edit table and breakpoints — Launch Lookup Table Editor dialog box**

button

Click this button to open the Lookup Table Editor. For more information, see “Edit Lookup Tables” in the Simulink documentation.

Clicking this button for a lookup table object lets you edit the object and save the new values for the object.

**Algorithm**

**Lookup method**

**Interpolation method — Method of interpolation between breakpoint values**

| Linear point-slope (default) | Flat | Nearest | Linear Lagrange | Cubic spline | Akima spline |

When an input falls between breakpoint values, the block interpolates the output value using neighboring breakpoints. For more information on interpolation methods, see “Interpolation Methods”.

**Dependencies**

- If you select **Cubic spline**, the block supports only scalar signals. The other interpolation methods support nonscalar signals.
- If you select **Akima spline**, the extrapolation method can only be **Akima spline**.

When set to the modified Akima interpolation method, this block does not support:

- Row-major array layout and algorithms optimized for row-major array layout
- Scaled double and fixed-point data types
- Simulink.LookupTable objects
- Code generation when the configuration parameter **Code Generation > Interface > Support non-finite numbers** check box is selected

**Programmatic Use**

**Block Parameter:** InterpMethod

**Type:** character vector

**Values:** 'Linear point-slope' | 'Flat' | 'Nearest' | 'Linear Lagrange' | 'Cubic spline' | 'Akima spline'
Default: 'Linear point-slope'

Extrapolation method — Method of handling input values that fall outside the range of a breakpoint data set
Clip (default) | Linear | Cubic spline | Akima spline

Select Clip, Linear, or Cubic spline. See “Extrapolation Methods” for more information.

If the extrapolation method is Linear, the extrapolation value is calculated based on the selected linear interpolation method. For example, if the interpolation method is linear Lagrange, the extrapolation method inherits the linear Lagrange equation to compute the extrapolated value.

Dependencies

• To select Cubic spline for Extrapolation method, you must also select Cubic spline for Interpolation method.
• To select Akima spline for Extrapolation method, you must also select Akima spline for Interpolation method.

Programmatic Use
Block Parameter: ExtrapMethod
Type: character vector
Values: 'Linear' | 'Clip' | 'Cubic spline' | 'Akima spline'
Default: 'Linear'

Index search method — Method of calculating table indices
Evenly spaced points (default) | Linear search | Binary search

Select Evenly spaced points, Linear search, or Binary search. Each search method has speed advantages in different circumstances:

• For evenly spaced breakpoint sets (for example, 10, 20, 30, and so on), you achieve optimal speed by selecting Evenly spaced points to calculate table indices.

This algorithm uses only the first two breakpoints of a set to determine the offset and spacing of the remaining points.

Note Set Index search method to Evenly spaced points when using the Simulink.LookupTable object to specify table data and the Breakpoints
Specification parameter of the referenced Simulink.LookupTable object is set to Even spacing.

- For unevenly spaced breakpoint sets, follow these guidelines:
  - If input signals do not vary much between time steps, selecting Linear search with Begin index search using previous index result produces the best performance.
  - If input signals jump more than one or two table intervals per time step, selecting Binary search produces the best performance.

A suboptimal choice of index search method can lead to slow performance of models that rely heavily on lookup tables.

**Note** The generated code stores only the first breakpoint, the spacing, and the number of breakpoints when:

- The breakpoint data is not tunable.
- The index search method is Evenly spaced points.

**Programmatic Use**

**Block Parameter:** IndexSearchMethod  
**Type:** character vector  
**Values:** 'Binary search' | 'Evenly spaced points' | 'Linear search'  
**Default:** 'Binary search'

**Begin index search using previous index result — Start using the index from the previous time step**

off (default) | on

Select this check box when you want the block to start its search using the index found at the previous time step. For inputs that change slowly with respect to the interval size, enabling this option can improve performance. Otherwise, the linear search and binary search methods can take longer, especially for large breakpoint sets.

**Dependencies**

To enable this parameter, set Index search method to Linear search or Binary search.
**Programmatic Use**

**Block Parameter:** BeginIndexSearchUsing PreviousIndexResult  
**Type:** character vector  
**Values:** 'off' | 'on'  
**Default:** 'off'

**Diagnostic for out-of-range input — Block action when input is out of range**

None (default) | Warning | Error

Specify whether to produce a warning or error when the input is out of range. Options include:

- **None** — Produce no response.
- **Warning** — Display a warning and continue the simulation.
- **Error** — Terminate the simulation and display an error.

**Programmatic Use**

**Block Parameter:** DiagnosticForOutOfRangeInput  
**Type:** character vector  
**Values:** 'None' | 'Warning' | 'Error'  
**Default:** 'None'

**Use last table value for inputs at or above last breakpoint — Method for computing output for inputs at or above last breakpoint**

off (default) | on

Using this check box, specify the indexing convention that the block uses to address the last element of a breakpoint set and its corresponding table value. This check box is relevant if the input is equal to or larger than the last element of the breakpoint data. Due to rounding, selecting and clearing this check box may result in differing results for the last breakpoint between simulation and code generation.

<table>
<thead>
<tr>
<th>Check Box</th>
<th>Index Used by Block</th>
<th>Interval Fraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selected</td>
<td>Last element of breakpoint data on the <strong>Table and Breakpoints</strong> tab</td>
<td>0</td>
</tr>
<tr>
<td>Cleared</td>
<td>Next-to-last element of breakpoint data on the <strong>Table and Breakpoints</strong> tab</td>
<td>1</td>
</tr>
</tbody>
</table>

Given an input \( u \) within range of a breakpoint set \( bp \), the interval fraction \( f \), in the range \( 0 \leq f \leq 1 \), is computed as shown below.
Suppose the breakpoint set is [1 4 5] and input \( u = 5.5 \). If you select this check box, the index is that of the last element (5) and the interval fraction is 0. If you clear this check box, the index is that of the next-to-last element (4) and the interval fraction is 1.

**Dependencies**

To enable this parameter, set:

- **Interpolation method** to Linear.
- **Extrapolation method** to Clip.

**Programmatic Use**

**Block Parameter:** UseLastTableValue  
**Type:** character vector  
**Values:** 'off' | 'on'  
**Default:** 'off'

**Input settings**

**Use one input port for all input data — Use only one input port**

off (default) | on

Select this check box to use only one input port that expects a signal that is \( n \) elements wide for an \( n \)-dimensional table. This option is useful for removing line clutter on a block diagram with many lookup tables.
Note When you select this check box, one input port with the label u appears on the block.

Programmatic Use
Block Parameter: UseOneInputPortForAllInputData
Type: character vector
Values: 'off' | 'on'
Default: 'off'

Code generation

Remove protection against out-of-range input in generated code —
Remove code that checks for out-of-range input values
off (default) | on

Specify whether or not to include code that checks for out-of-range input values.

<table>
<thead>
<tr>
<th>Check Box</th>
<th>Result</th>
<th>When to Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>on</td>
<td>Generated code does not include conditional statements to check for out-of-range breakpoint inputs. When the input is out-of-range, it may cause undefined behavior for generated code and simulations using accelerator mode.</td>
<td>For code efficiency</td>
</tr>
<tr>
<td>off</td>
<td>Generated code includes conditional statements to check for out-of-range inputs.</td>
<td>For safety-critical applications</td>
</tr>
</tbody>
</table>

If your input is not out of range, you can select the **Remove protection against out-of-range index in generated code** check box for code efficiency. By default, this check box is cleared. For safety-critical applications, do not select this check box. If you want to select the **Remove protection against out-of-range index in generated code** check box, first check that your model inputs are in range. For example:
1. Clear the **Remove protection against out-of-range index in generated code** check box.

2. Set the **Diagnostic for out-of-range input** parameter to **Error**.

3. Simulate the model in normal mode.

4. If there are out-of-range errors, fix them to be in range and run the simulation again.

5. When the simulation no longer generates out-of-range input errors, select the **Remove protection against out-of-range index in generated code** check box.

**Note** When you select the **Remove protection against out-of-range index in generated code** check box and the input \( k \) or \( f \) is out of range, the behavior is undefined for generated code and simulations using accelerator mode.

Depending on your application, you can run the following Model Advisor checks to verify the usage of this check box:

- **By Product > Embedded Coder > Identify lookup table blocks that generate expensive out-of-range checking code**
- **By Product > Simulink Check > Modeling Standards > DO-178C/DO-331 Checks > Check usage of lookup table blocks**

For more information about the Model Advisor, see “Run Model Advisor Checks”.

**Programmatic Use**

**Block Parameter:** RemoveProtectionInput  
**Type:** character vector  
**Values:** 'off' | 'on'  
**Default:** 'off'

**Support tunable table size in code generation — Enable tunable table size in the generated code**

**off** (default) | **on**

Select this check box to enable tunable table size in the generated code. This option enables you to change the size and values of the lookup table and breakpoint data in the generated code without regenerating or recompiling the code. You can only decrease the size of the lookup table and breakpoint data.

**Dependencies**

If you set **Interpolation method** to **Cubic spline**, this check box is not available.
**Programmatic Use**

**Block Parameter:** SupportTunableTableSize  
**Type:** character vector  
**Values:** 'off' | 'on'  
**Default:** 'off'

**Sample time — Specify sample time as a value other than -1**

-1 (default) | scalar | vector  

Specify the sample time as a value other than -1. For more information, see “Specify Sample Time”.

**Dependencies**

This parameter is not visible unless it is explicitly set to a value other than -1. To learn more, see “Blocks for Which Sample Time Is Not Recommended”.

**Programmatic Use**

**Block Parameter:** SampleTime  
**Type:** character vector  
**Values:** scalar or vector  
**Default:** '-1'

**Maximum indices for each dimension — Maximum index value for each table dimension**

[] (default) | scalar or vector of positive integer values  

Example: [4 6] for a 5-by-7 table

Specify the maximum index values for each table dimension using zero-based indexing. You can specify a scalar or vector of positive integer values using the following data types:

- Built-in floating-point types: double and single
- Built-in integer types: int8, int16, int32, uint8, uint16, and uint32

Examples of valid specifications include:

- [4 6] for a 5-by-7 table
- [int8(2) int16(5) int32(9)] for a 3-by-6-by-10 table
- A Simulink.Parameter whose value on generating code is one less than the dimensions of the table data. For more information, see “Tunable Table Size in the Generated Code” on page 1-1202.
Dependencies

To enable this parameter, select **Support tunable table size in code generation**. On tuning this parameter in the generated code, provide the new table data and breakpoints along with the tuned parameter value.

**Programmatic Use**

- **Block Parameter:** MaximumIndicesForEachDimension
  - **Type:** character vector
  - **Values:** scalar or vector of positive integer values
  - **Default:** '[]'

**Data Types**

**Table data — Data type of table data**

Specify the table data type. You can set it to:

- A rule that inherits a data type, for example, *Inherit: Same as output*
- The name of a built-in data type, for example, *single*
- The name of a data type object, for example, a Simulink.NumericType object
- An expression that evaluates to a data type, for example, *fixdt(1,16,0)*

Click the **Show data type assistant** button to display the **Data Type Assistant**, which helps you set the data type attributes. For more information, see “Specify Data Types Using Data Type Assistant”.

**Tip** Specify a table data type different from the output data type for these cases:

- Lower memory requirement for storing table data that uses a smaller type than the output signal
- Sharing of prescaled table data between two n-D Lookup Table blocks with different output data types
- Sharing of custom storage table data in the generated code for blocks with different output data types
**Programmatic Use**

**Block Parameter:** TableDataTypeStr  
**Type:** character vector  
**Values:**  
- 'Inherit: Inherit from 'Table data' '  
- 'Inherit: Same as output'  
- 'double'  
- 'single'  
- 'int8'  
- 'uint8'  
- 'int16'  
- 'uint16'  
- 'int32'  
- 'uint32'  
- 'int64'  
- 'uint64'  
- 'fixdt(1,16)'  
- 'fixdt(1,16,0)'  
- 'fixdt(1,16,2^0,0)'  
- '<data type expression>'  
**Default:** 'Inherit: Same as output'

**Table data Minimum — Minimum value of the table data**  
[ ] | scalar

Specify the minimum value for table data. The default value is [ ] (unspecified).

**Programmatic Use**

**Block Parameter:** TableMin  
**Type:** character vector  
**Values:** scalar  
**Default:** '[]'

**Table data Maximum — Maximum value of the table data**  
[ ] | scalar

Specify the maximum value for table data. The default value is [ ] (unspecified).

**Programmatic Use**

**Block Parameter:** TableMax  
**Type:** character vector  
**Values:** scalar  
**Default:** '[]'

**Breakpoints — Breakpoint data type**

Inherit: Same as corresponding input (default) | double | single | int8 | uint8 | int16 | uint16 | int32 | uint32 | int64 | uint64 | fixdt(1,16) | fixdt(1,16,0) | fixdt(1,16,2^0,0) | Enum: <class name> | <data type expression>

Specify the data type for a set of breakpoint data. You can set it to:

- A rule that inherits a data type, for example, Inherit: Same as corresponding input
- The name of a built-in data type, for example, single
• The name of a data type class, for example, an enumerated data type class
• The name of a data type object, for example, a Simulink.NumericType object
• An expression that evaluates to a data type, for example, fixdt(1,16,0)

**Tip**

- Breakpoints support unordered enumerated data. As a result, linear searches are also unordered, which offers flexibility but can impact performance. The search begins from the first element in the breakpoint.
- If the **Begin index search using previous index result** check box is selected, you must use ordered monotonically increasing data. This ordering improves performance.
- For enumerated data, **Extrapolation method** must be Clip.
- The block does not support out-of-range input for enumerated data. When specifying enumerated data, include the entire enumeration set in the breakpoint data set. For example, use the `enumeration` function.

This is a limitation for using enumerated data with this block:

- The block does not support out-of-range input for enumerated data. When specifying enumerated data, include the entire enumeration set in the breakpoint data set. For example, use the `enumeration` function.

Click the **Show data type assistant** button to display the **Data Type Assistant**, which helps you set the data type attributes. For more information, see “Specify Data Types Using Data Type Assistant”.

**Tip** Specify a breakpoint data type different from the corresponding input data type for these cases:

- Lower memory requirement for storing breakpoint data that uses a smaller type than the input signal
- Sharing of prescaled breakpoint data between two n-D Lookup Table blocks with different input data types
- Sharing of custom storage breakpoint data in the generated code for blocks with different input data types
Programmatic Use

**Block Parameter:** BreakpointsForDimension1DataTypeStr | BreakpointsForDimension2DataTypeStr| ... | BreakpointsForDimension30DataTypeStr

**Type:** character vector

**Values:** 'Inherit: Same as corresponding input' | 'Inherit: Inherit from 'Breakpoint data'' | 'double' | 'single' | 'int8' | 'uint8' | 'int16' | 'uint16' | 'int32' | 'uint32' | 'int64' | 'uint64' | 'fixdt(1,16)' | 'fixdt(1,16,0)' | 'fixdt(1,16,2^0,0)'</data type expression>

**Default:** 'Inherit: Same as corresponding input'

Breakpoints Minimum — Minimum value breakpoint data can have

```
[] | scalar
```

Specify the minimum value that a set of breakpoint data can have. The default value is [] (unspecified).

Programmatic Use

**Block Parameter:** BreakpointsForDimension1Min | BreakpointsForDimension2Min | ... | BreakpointsForDimension30Min

**Type:** character vector

**Values:** scalar

**Default:** '[]'

Breakpoints Maximum — Maximum value breakpoint data can have

```
[] | scalar
```

Specify the maximum value that a set of breakpoint data can have. The default value is [] (unspecified).

Programmatic Use

**Block Parameter:** BreakpointsForDimension1Max | BreakpointsForDimension2Max | ... | BreakpointsForDimension30Max

**Type:** character vector

**Values:** scalar

**Default:** '[]'

**Fraction — Fraction data type**

Inherit: Inherit via internal rule (default) | double | single | fixdt(1,16,0) | <data type expression>

Specify the fraction data type. You can set it to:
• A rule that inherits a data type, for example, Inherit: Inherit via internal rule
• The name of a built-in data type, for example, single
• The name of a data type object, for example, a Simulink.NumericType object
• An expression that evaluates to a data type, for example, fixdt(1,16,0)

Click the Show data type assistant button to display the Data Type Assistant, which helps you set the data type attributes. For more information, see “Specify Data Types Using Data Type Assistant”.

Programmatic Use
Block Parameter: FractionDataTypeStr
Type: character vector
Values: 'Inherit: Inherit via internal rule' | 'double' | 'single' | 'fixdt(1,16,0)' | '<data type expression>'
Default: 'Inherit: Inherit via internal rule'

Intermediate results — Intermediate results data type
Inherit: Same as output (default) | double | single | int8 | uint8 | int16 | uint16 | int32 | uint32 | int64 | uint64 | fixdt(1,16) | fixdt(1,16,0) | fixdt(1,16,2^0,0) | <data type expression>

Specify the intermediate results data type. You can set it to:
• A rule that inherits a data type, for example, Inherit: Same as output
• The name of a built-in data type, for example, single
• The name of a data type object, for example, a Simulink.NumericType object
• An expression that evaluates to a data type, for example, fixdt(1,16,0)

Click the Show data type assistant button to display the Data Type Assistant, which helps you set the data type attributes. For more information, see “Specify Data Types Using Data Type Assistant”.

Tip Use this parameter to specify higher (or lower) precision for internal computations than for table data or output data.
**Programmatic Use**

**Block Parameter:** IntermediateResultsDataTypeStr  
**Type:** character vector  
**Values:**  
- 'Inherit: Inherit via internal rule' | 'Inherit: Same as output' | 'double' | 'single' | 'int8' | 'uint8' | 'int16' | 'uint16' | 'int32' | 'uint32' | 'int64' | 'uint64' | 'fixdt(1,16,0)' | 'fixdt(1,16,2^0,0)' | '<data type expression>'  
**Default:** 'Inherit: Same as output'

**Output — Output data type**

Inherit: Same as input (default) | double | single | int8 | uint8 | int16 | uint16 | int32 | uint32 | int64 | uint64 | fixdt(1,16) | fixdt(1,16,0) | fixdt(1,16,2^0,0) | '<data type expression>'

Specify the output data type. You can set it to:

- A rule that inherits a data type, for example, Inherit: Inherit via back propagation
- The name of a built-in data type, for example, single
- The name of a data type object, for example, a Simulink.NumericType object
- An expression that evaluates to a data type, for example, fixdt(1,16,0)

Click the **Show data type assistant** button to display the **Data Type Assistant**, which helps you set the data type attributes. For more information, see “Specify Data Types Using Data Type Assistant”.

**Programmatic Use**

**Block Parameter:** OutDataTypeStr  
**Type:** character vector  
**Values:**  
- 'Inherit: Inherit via back propagation' | 'Inherit: Inherit from table data' | 'Inherit: Same as first input' | 'double' | 'single' | 'int8' | 'uint8' | 'int16' | 'uint16' | 'int32' | 'uint32' | 'int64' | 'uint64' | 'fixdt(1,16,0)' | 'fixdt(1,16,2^0,0)' | '<data type expression>'  
**Default:** 'Inherit: Same as first input'

**Output Minimum — Minimum value the block can output**

[] | scalar
Specify the minimum value that the block outputs. The default value is [ ] (unspecified). Simulink software uses this value to perform:

- Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters”).
- Simulation range checking (see “Signal Ranges”).
- Automatic scaling of fixed-point data types.
- Optimization of the code that you generate from the model. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. For more information, see “Optimize using the specified minimum and maximum values” (Simulink Coder).

**Programmatic Use**

**Block Parameter:** OutMin  
**Type:** character vector  
**Values:** scalar  
**Default:** ' [] '

**Output Maximum — Maximum value the block can output**  
[] | scalar

Specify the maximum value that the block can output. The default value is [ ] (unspecified). Simulink software uses this value to perform:

- Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters”).
- Simulation range checking (see “Signal Ranges”).
- Automatic scaling of fixed-point data types.
- Optimization of the code that you generate from the model. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. For more information, see “Optimize using the specified minimum and maximum values” (Simulink Coder).

**Programmatic Use**

**Block Parameter:** OutMax  
**Type:** character vector  
**Values:** scalar  
**Default:** ' [] '
**Internal rule priority — Internal rule for intermediate calculations**
Speed (default) | Precision

Specify the internal rule for intermediate calculations. Select Speed for faster calculations. If you do, a loss of accuracy might occur, usually up to 2 bits.

**Programmatic Use**
**Block Parameter:** InternalRulePriority
**Type:** character vector
**Values:** 'Speed' | 'Precision'
**Default:** 'Speed'

**Require all inputs to have the same data type — Require all inputs to have the same data type**
on (default) | off

Select to require all inputs to have the same data type.

**Programmatic Use**
**Block Parameter:** InputSameDT
**Type:** character vector
**Values:** 'off' | 'on'
**Default:** 'on'

**Lock data type settings against changes by the fixed-point tools — Prevent fixed-point tools from overriding data types**
off (default) | on

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on this block. For more information, see “Lock the Output Data Type Setting” (Fixed-Point Designer).

**Programmatic Use**
**Block Parameter:** LockScale
**Type:** character vector
**Values:** 'off' | 'on'
**Default:** 'off'

**Integer rounding mode — Rounding mode for fixed-point operations**
Simplest (default) | Ceiling | Convergent | Floor | Nearest | Round | Zero
Specify the rounding mode for fixed-point lookup table calculations that occur during simulation or execution of code generated from the model. For more information, see “Rounding” (Fixed-Point Designer).

This option does not affect rounding of values of block parameters. Simulink rounds such values to the nearest representable integer value. To control the rounding of a block parameter, enter an expression using a MATLAB rounding function into the edit field on the block dialog box.

**Programmatic Use**

**Block Parameter:** RndMeth  
**Type:** character vector  
**Values:** 'Ceiling' | 'Convergent' | 'Floor' | 'Nearest' | 'Round' | 'Simplest' | 'Zero'  
**Default:** 'Simplest'

**Saturate on integer overflow — Method of overflow action**

<table>
<thead>
<tr>
<th>Action</th>
<th>Reasons for Taking This Action</th>
<th>What Happens for Overflows</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Select this check box (on).</td>
<td>Your model has possible overflow and you want explicit saturation protection in the generated code.</td>
<td>Overflows saturate to either the minimum or maximum value that the data type can represent.</td>
<td>An overflow associated with a signed 8-bit integer can saturate to -128 or 127.</td>
</tr>
<tr>
<td>Do not select this check box (off).</td>
<td>You want to optimize efficiency of your generated code. You want to avoid overspecifying how a block handles out-of-range signals. For more information, see “Check for Signal Range Errors”.</td>
<td>Overflows wrap to the appropriate value that is representable by the data type.</td>
<td>The number 130 does not fit in a signed 8-bit integer and wraps to -126.</td>
</tr>
</tbody>
</table>

**Tip** If you save your model as version R2009a or earlier, this check box setting has no effect and no saturation code appears. This behavior preserves backward compatibility.
When you select this check box, saturation applies to every internal operation on the block, not just the output or result. In general, the code generation process can detect when overflow is not possible. In this case, the code generator does not produce saturation code.

**Programmatic Use**

**Block Parameter:** SaturateOnIntegerOverflow  
**Type:** character vector  
**Values:** 'off' | 'on'  
**Default:** 'off'

### Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>double</th>
<th>enumerated</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### More About

**Tunable Table Size in the Generated Code**

Suppose that you have a lookup table and want to make the size tunable in the generated code. When you use Simulink.LookupTable and Simulink.Breakpoint objects to configure lookup table data for calibration in the generated code, use the SupportTunableSize property of the objects to enable a tunable table size. When you do not use these classes, use the **Support tunable table size in code generation** parameter in an n-D Lookup Table block to enable a tunable table size.

Assume that:
• You define a `Simulink.Parameter` structure in the preload function of your model:

```matlab
p = Simulink.Parameter;
p.Value.MaxIdx = [2 2];
p.Value.BP1 = [1 2 3];
p.Value.BP2 = [1 4 16];
p.Value.Table = [4 5 6; 16 19 20; 10 18 23];
p.DataType = 'Bus: slLookupTable';
p.CoderInfo.StorageClass = 'ExportedGlobal';

% Create bus object slBus1 from MATLAB structure
Simulink.Bus.createObject(p.Value);
slLookupTable = slBus1;
slLookupTable.Elements(1).DataType = 'uint32';
```

• These block parameters apply in the n-D Lookup Table block.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of table dimensions</td>
<td>2</td>
</tr>
<tr>
<td>Table data</td>
<td><code>p.Table</code></td>
</tr>
<tr>
<td>Breakpoints 1</td>
<td><code>p.BP1</code></td>
</tr>
<tr>
<td>Breakpoints 2</td>
<td><code>p.BP2</code></td>
</tr>
<tr>
<td>Support tunable table size in code generation</td>
<td>on</td>
</tr>
<tr>
<td>Maximum indices for each dimension</td>
<td><code>p.MaxIdx</code></td>
</tr>
</tbody>
</table>

The generated `model_types.h` header file contains a type definition that looks something like this.

```c
typedef struct {
    uint32_T MaxIdx[2];
    real_T BP1[3];
    real_T BP2[3];
    real_T Table[9];
} slLookupTable;
```

The generated `model.c` file contains code that looks something like this.

```c
/* Exported block parameters */
slLookupTable p = {
    { 2U, 2U },
```
The highlighted line of code specifies a tunable table size for the lookup table. You can change the size and values of the lookup table and breakpoint data without regenerating or recompiling the code.

**Enumerated Values in Lookup Tables**

Suppose that you have a lookup table with an enumerated class like this defined:

```plaintext
classdef(Enumeration) Gears < Simulink.IntEnumType
    enumeration
        GEAR1(1),
        GEAR2(2),
        GEAR3(4),
        GEAR4(8),
        SPORTS(16),
        REVERSE(-1),
```
n-D Lookup Table has these settings:

- **Number of dimensions** to 1.
- **Table data** value is [5 10 20 40 80 -5 0].
- **Breakpoints 1** value is enumeration('Gears').
- Interpolation method is Flat.
- For an unordered search, set **Index search method** to Linear search and clear the **Begin index search using previous index result** check box.

Simulation produces a vector [10 -5 80], which correspond to GEAR2, REVERSE, and SPORTS.

**Extended Capabilities**

**C/C++ Code Generation**
Generate C and C++ code using Simulink® Coder™.

**HDL Code Generation**
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.
**HDL Architecture**

This block has a single, default HDL architecture.

**HDL Block Properties**

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConstrainedOutput Pipeline</td>
<td>Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. For more details, see “ConstrainedOutputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td>InputPipeline</td>
<td>Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “InputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td>OutputPipeline</td>
<td>Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “OutputPipeline” (HDL Coder).</td>
</tr>
</tbody>
</table>

**Complex Data Support**

This block supports code generation for complex signals.

**Restrictions**

**MAX 10 Device Settings**

If you use Intel MAX 10 device, to map the lookup table to RAM, add this Tcl command when creating the project in the Quartus tool:

```
set_global_assignment -name INTERNAL_FLASH_UPDATE_MODE "SINGLE IMAGE WITH ERM"
```

**Required Block Settings**
### Block Setting

<table>
<thead>
<tr>
<th>Block Setting</th>
<th>HDL Coder support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of table dimensions</td>
<td>Specify up to a maximum dimension of 2.</td>
</tr>
<tr>
<td>Breakpoints specification</td>
<td>Select either Explicit values or Even spacing.</td>
</tr>
<tr>
<td>Index search method</td>
<td>Select Evenly spaced points.</td>
</tr>
<tr>
<td>Extrapolation method</td>
<td>Select Clip. The code generator does not support extrapolation beyond even bounds.</td>
</tr>
<tr>
<td>Interpolation method</td>
<td>Select Flat or Linear point-slope.</td>
</tr>
<tr>
<td>Diagnostic for out-of-range input</td>
<td>Select Error. If you specify other options, HDL Coder generates a warning.</td>
</tr>
<tr>
<td>Use last table value for inputs at or above last breakpoint</td>
<td>Select this check box.</td>
</tr>
<tr>
<td>Require all inputs to have the same data type</td>
<td>Select this check box.</td>
</tr>
<tr>
<td>Fraction</td>
<td>Select Inherit: Inherit via internal rule.</td>
</tr>
<tr>
<td>Integer rounding mode</td>
<td>Select Zero, Floor, or Simplest.</td>
</tr>
</tbody>
</table>

### Avoid Generation of Divide Operator

If HDL Coder encounters conditions under which a division operation is required to match the model simulation behavior, a warning is displayed. The conditions described cause this block to emit a divide operator. When you use this block for HDL code generation, avoid the following conditions:

- If the block is configured to use interpolation, a division operator is required. To avoid this requirement, set **Interpolation method** to Flat.
- Uneven table spacing. HDL code generation requires the block to use the evenly spaced points algorithm. The block mapping from the input data type to the zero-based table index in general requires a division. When the breakpoint spacing is an exact power of 2, this divide is implemented as a shift instead of as a divide. To adjust the breakpoint spacing, adjust the number of breakpoints in the table, or the difference between the left and right bounds of the breakpoint range.

### Table Data Typing and Sizing
• It is good practice to structure your table such that the spacing between breakpoints is a power of two. If the breakpoint spacing does not meet this condition, HDL Coder issues a warning. When the breakpoint spacing is a power of two, you can replace division operations in the prelookup step with right-shift operations.
• Table data must resolve to a nonfloating-point data type.
• All ports on the block require scalar values.

**PLC Code Generation**
Generate Structured Text code using Simulink® PLC Coder™.

Simulink PLC Coder™ has limited support for lookup table blocks. The coder does not support:

• Number of dimensions greater than 2
• Cubic spline interpolation method
• Begin index search using a previous index mode
• Cubic spline extrapolation method

**Fixed-Point Conversion**
Design and simulate fixed-point systems using Fixed-Point Designer™.

**See Also**
Direct Lookup Table (n-D) | Interpolation Using Prelookup | Lookup Table Dynamic | Prelookup | Simulink.Breakpoint | Simulink.LookupTable

**Topics**
“Import Lookup Table Data from MATLAB”
“About Lookup Table Blocks”
“Anatomy of a Lookup Table”
“Enter Breakpoints and Table Data”
“Guidelines for Choosing a Lookup Table”
“Interpolation Algorithm for Row-Major Array Layout” (Simulink Coder)
Introduced in R2011a
Lookup Table Dynamic

Approximate a one-dimensional function using dynamic table

Library: Simulink / Lookup Tables

Description

How This Block Differs from Other Lookup Table Blocks

The Lookup Table Dynamic block computes an approximation of a function \( y = f(x) \) using \( x_{\text{dat}} \) and \( y_{\text{dat}} \) vectors. The lookup method can use interpolation, extrapolation, or the original values of the input.

Using the Lookup Table Dynamic block, you can change the table data without stopping the simulation. For example, you can incorporate new table data if the physical system you are simulating changes.

Inputs for Breakpoint and Table Data

The \( x_{\text{dat}} \) vector is the breakpoint data, which must be strictly monotonically increasing. The value of the next element in the vector must be greater than the value of the preceding element after conversion to a fixed-point data type. Due to quantization, \( x_{\text{dat}} \) can be strictly monotonic for a floating-point data type, but not after conversion to a fixed-point data type.

The \( y_{\text{dat}} \) vector is the table data, which is an evaluation of the function at the breakpoint values.

Note The inputs to \( x_{\text{dat}} \) and \( y_{\text{dat}} \) cannot be scalar (one-element array) values. If you provide a scalar value to either of these inputs, you see an error upon simulation. Provide a 1-by-n vector to both the \( x_{\text{dat}} \) and \( y_{\text{dat}} \) inputs.
Lookup Table Definition

You define the lookup table by feeding xdat and ydat as 1-by-n vectors to the block. To reduce ROM usage in the generated code for this block, you can use different data types for xdat and ydat. However, these restrictions apply:

- The xdat breakpoint data and the x input vector must have the same sign, bias, and fractional slope. Also, the precision and range for x must be greater than or equal to the precision and range for xdat.
- The ydat table data and the y output vector must have the same sign, bias, and fractional slope.

Tip Breakpoints with even spacing can make Simulink Coder generated code division-free. For more information, see fixpt_evenspace_cleanup in the Simulink documentation and “Identify questionable fixed-point operations” (Embedded Coder) in the Simulink Coder documentation.

How the Block Generates Output

The block uses the input values to generate output using the method you select for Lookup Method:

<table>
<thead>
<tr>
<th>Lookup Method</th>
<th>Block Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interpolation-Extrapolation</td>
<td>Performs linear interpolation and extrapolation of the inputs.</td>
</tr>
<tr>
<td></td>
<td>• If the input matches a breakpoint, the output is the corresponding element in the table data.</td>
</tr>
<tr>
<td></td>
<td>• If the input does not match a breakpoint, the block performs linear interpolation between two elements of the table to determine the output. If the input falls outside the range of breakpoint values, the block extrapolates using the first two or last two points.</td>
</tr>
</tbody>
</table>

Note If you select this lookup method, Simulink Coder software cannot generate code for this block.
### Lookup Method

<table>
<thead>
<tr>
<th>Lookup Method</th>
<th>Block Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interpolation-Use End Values (default)</td>
<td>Performs linear interpolation but does not extrapolate outside the end points of the breakpoint data. Instead, the block uses the end values.</td>
</tr>
<tr>
<td>Use Input Nearest</td>
<td>Finds the element in xdat nearest the current input. The corresponding element in ydat is the output.</td>
</tr>
<tr>
<td>Use Input Below</td>
<td>Finds the element in xdat nearest and below the current input. The corresponding element in ydat is the output. If there is no element in xdat below the current input, the block finds the nearest element.</td>
</tr>
<tr>
<td>Use Input Above</td>
<td>Finds the element in xdat nearest and above the current input. The corresponding element in ydat is the output. If there is no element in xdat above the current input, the block finds the nearest element.</td>
</tr>
</tbody>
</table>

**Note** The Use Input Nearest, Use Input Below, and Use Input Above methods perform the same action when the input \( x \) matches a breakpoint value.

Some continuous solvers subdivide the simulation time span into major and minor time steps. A minor time step is a subdivision of the major time step. The solver produces a result at each major time step and uses results at minor time steps to improve the accuracy of the result at the major time step. For continuous solvers, the output of the Lookup Table Dynamic block can appear like a stair step because the signal is fixed in minor time step to avoid incorrect results. For more information about the effect of solvers on block output, see “Compare Solvers” in the Simulink documentation.

### Ports

**Input**

\[ x \rightarrow \text{input vector} \]

*scalar|vector|2-D array*

The block accepts real-valued or complex-valued inputs.

Example: 2:12
Dependencies

The x input vector and the xdat breakpoint data must have the same sign, bias, and fractional slope. Also, the precision and range for x must be greater than or equal to the precision and range for xdat.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | bus

xdat — breakpoint data
1-by-n vector of strictly monotonically increasing values

The xdat vector is the breakpoint data, which must be strictly monotonically increasing. The value of the next element in the vector must be greater than the value of the preceding element after conversion to a fixed-point data type. Due to quantization, xdat can be strictly monotonic for a floating-point data type, but not after conversion to a fixed-point data type.

Tip Breakpoints with even spacing can make Simulink Coder generated code division-free. For more information, see fixpt_evenspace_cleanup in the Simulink documentation and “Identify questionable fixed-point operations” (Embedded Coder) in the Simulink Coder documentation.

Example: 1:10

Dependencies

The xdat breakpoint data and the x input vector must have the same sign, bias, and fractional slope. Also, the precision and range for x must be greater than or equal to the precision and range for xdat.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | bus

ydat — table data
1-by-n vector

The ydat input is a 1-by-n vector of real-valued or complex-valued table data, which is an evaluation of the function at the breakpoint values.

Example: [0 3 12 27 48 75 108 147 192 243 300]
Dependencies

The \textit{ydat} table data and the \textit{y} output vector must have the same sign, bias, and fractional slope.

Data Types: \textit{single} | \textit{double} | \textit{int8} | \textit{int16} | \textit{int32} | \textit{uint8} | \textit{uint16} | \textit{uint32} | \textit{Boolean} | \textit{fixed point} | bus

Output

\( y \) — \textit{Approximation of } \( y = f(x) \) \textit{using dynamic table data}

1-by-\textit{n} vector

The block computes an approximation of a function \( y = f(x) \) using the \textit{xdat} and \textit{ydat} input vectors. The lookup method can use interpolation, extrapolation, or the original values of the input.

Dependencies

The \textit{ydat} table data and the \textit{y} output vector must have the same sign, bias, and fractional slope.

Data Types: \textit{single} | \textit{double} | \textit{int8} | \textit{int16} | \textit{int32} | \textit{uint8} | \textit{uint16} | \textit{uint32} | \textit{Boolean} | \textit{fixed point}

Parameters

Main Tab

\textbf{Lookup Method — Specify the lookup method}

Interpolation-Use End Values (default) | Interpolation-Extrapolation | Use Input Nearest | Use Input Below | Use Input Above

The block computes output by applying the \textbf{Lookup Method} you select to the input vectors of breakpoint data (\textit{xdat}) and table data (\textit{ydat}). For details, see “How the Block Generates Output” on page 1-1211.

\textbf{Programmatic Use}

\textbf{Block Parameter:} LookUpMeth

\textbf{Type:} character vector
Values: 'Interpolation-Extrapolation' | 'Interpolation-Use End Values' | 'Use Input Nearest' | 'Use Input Below' | 'Use Input Above'
Default: 'Interpolation-Use End Values'

Signal Attributes Tab

**Output data type — Output data type**

Specify the data type of the output signal \( y \).

Click the **Show data type assistant** button to display the **Data Type Assistant**, which helps you set the data type attributes. For more information, see “Specify Data Types Using Data Type Assistant”.

**Dependencies**

The \( y \) data table and the \( y \) output vector must have the same sign, bias, and fractional slope.

**Programmatic Use**

**Block Parameter:** OutDataTypeStr

**Type:** character vector

**Values:** 'Inherit: Inherit via back propagation' | 'double' | 'single' | 'int8' | 'uint8' | 'int16' | 'uint16' | 'int32' | 'uint32' | 'int64' | 'uint64' | 'boolean' | 'fixdt(1,16,0)' | 'fixdt(1,16,2^0,0)' | <data type expression>

**Default:** 'double'

**Lock output data type setting against changes by the fixed-point tools — Prevent fixed-point tools from overriding Output data type**

Select this parameter to prevent the fixed-point tools from overriding the **Output** data type you specify on the block. For more information, see “Use Lock Output Data Type Setting” (Fixed-Point Designer).

**Programmatic Use**

**Block Parameter:** LockScale
Type: character vector
Values: 'off' | 'on'
Default: 'off'

Integer rounding mode — Rounding mode for fixed-point operations
Floor (default) | Ceiling | Convergent | Nearest | Round | Simplest | Zero

Specify the rounding mode for fixed-point operations. For more information, see “Rounding” (Fixed-Point Designer).

Block parameters always round to the nearest representable value. To control the rounding of a block parameter, enter an expression using a MATLAB rounding function into the mask field.

Programmatic Use
Block Parameter: RndMeth
Type: character vector
Values: 'Ceiling' | 'Convergent' | 'Floor' | 'Nearest' | 'Round' | 'Simplest' | 'Zero'
Default: 'Floor'

Saturate to max or min when overflows occur — Method of overflow action
off (default) | on

When you select this check box, overflows saturate to the maximum or minimum value that the data type can represent. Otherwise, overflows wrap.

When you select this check box, saturation applies to every internal operation on the block, not just the output or result. In general, the code generation process can detect when overflow is not possible. In this case, the code generator does not produce saturation code.

Programmatic Use
Block Parameter: DoSatur
Type: character vector
Values: 'off' | 'on'
Default: 'off'
**Block Characteristics**

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>double</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Direct Feedthrough</strong></td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Multidimensional Signals</strong></td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Variable-Size Signals</strong></td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Zero-Crossing Detection</strong></td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Extended Capabilities**

**C/C++ Code Generation**
Generate C and C++ code using Simulink® Coder™.

**PLC Code Generation**
Generate Structured Text code using Simulink® PLC Coder™.

The Simulink PLC Coder software does not support the Simulink Lookup Table Dynamic block. For your convenience, the plclib/Simulink/Lookup Tables library contains an implementation of a dynamic table lookup block using the Prelookup and Interpolation Using Prelookup blocks.

**Fixed-Point Conversion**
Design and simulate fixed-point systems using Fixed-Point Designer™.

**See Also**
n-D Lookup Table
Topics

“Entering Data Using Inports of the Lookup Table Dynamic Block”
“Nonlinearity”
“About Lookup Table Blocks”
“Anatomy of a Lookup Table”
“Guidelines for Choosing a Lookup Table”

Introduced before R2006a
Magnitude-Angle to Complex

Convert magnitude and/or a phase angle signal to complex signal

Library: 
- Simulink / Math Operations
- HDL Coder / HDL Floating Point Operations

Description

Supported Operations

The Magnitude-Angle to Complex block converts magnitude and phase angle inputs to a complex output. The angle input must be in rad.

The block supports the following combinations of input dimensions when there are two block inputs:

- Two inputs of equal dimensions
- One scalar input and the other an n-dimensional array

If the block input is an array, the output is an array of complex signals. The elements of a magnitude input vector map to the magnitudes of the corresponding complex output elements. Similarly, the elements of an angle input vector map to the angles of the corresponding complex output elements. If one input is a scalar, it maps to the corresponding component (magnitude or angle) of all the complex output signals.

Effect of Out-of-Range Input on CORDIC Approximations

If you use the CORDIC approximation method [1], the block input for phase angle has the following restrictions:

- For signed fixed-point types, the input angle must fall within the range \([-2\pi, 2\pi)\) rad.
- For unsigned fixed-point types, the input angle must fall within the range \([0, 2\pi)\) rad.

The following table summarizes what happens for an out-of-range input:
<table>
<thead>
<tr>
<th>Block Usage</th>
<th>Effect of Out-of-Range Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simulation</td>
<td>An error appears.</td>
</tr>
<tr>
<td>Generated code</td>
<td>Undefined behavior occurs.</td>
</tr>
<tr>
<td>Accelerator modes</td>
<td></td>
</tr>
</tbody>
</table>

Ensure that you use an in-range input for the Magnitude-Angle to Complex block when you use the CORDIC approximation. Avoid relying on undefined behavior for generated code or accelerator modes.

**Ports**

**Input**

<table>
<thead>
<tr>
<th>u</th>
<th>— Magnitude</th>
</tr>
</thead>
<tbody>
<tr>
<td>scalar</td>
<td>vector</td>
</tr>
</tbody>
</table>

Magnitude, specified as a real-valued scalar, vector, or matrix.

**Dependencies**

- To enable this port, set **Input** to Magnitude and angle.

**Limitations**

- If one input has a floating-point data type, the other input must use the same data type. For example, both signals must be double or single.
- Fixed-point data types are supported only when you set the **Approximation method** to CORDIC. When one input has a fixed-point data type, the other input must also have a fixed-point data type.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

<table>
<thead>
<tr>
<th>u</th>
<th>— Radian phase angle</th>
</tr>
</thead>
<tbody>
<tr>
<td>scalar</td>
<td>vector</td>
</tr>
</tbody>
</table>

Radian phase angle, specified as a real-valued scalar, vector, or matrix. To compute the CORDIC approximation, the input angle must be between:
• \([-2\pi, 2\pi)\) rad, for signed fixed-point types
• \([0, 2\pi)\) rad, for unsigned fixed-point types

For more information, see “Effect of Out-of-Range Input on CORDIC Approximations” on page 1-1219.

**Dependencies**

• To enable this port, set **Input** to **Magnitude** and **angle**.

**Limitations**

• If one input has a floating-point data type, the other input must use the same data type. For example, both signals must be double or single.
• Fixed-point data types are supported only when you set the **Approximation method** to CORDIC. If one input has a fixed-point data type, the other input must also have a fixed-point data type.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

**Port_1 — Magnitude or radian phase angle**
scalar | vector | matrix

Magnitude, or radian phase angle, specified as a real-valued scalar, vector, or matrix.

• When you set **Input** to **Magnitude**, you specify the magnitude at the input port, and the angle on the dialog box.
• When you set **Input** to **Angle**, you specify the angle at the input port, and the magnitude on the dialog box.

**Dependencies**

To enable this port, set **Input** to **Magnitude** or **Angle**.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

**Output**

**Port_1 — Complex signal**
scalar | vector | matrix
Complex signal, formed from the magnitude and phase angle you specify.

If the block input is an array, the output is an array of complex signals. The elements of a magnitude input vector map to the magnitudes of the corresponding complex output elements. Similarly, the elements of an angle input vector map to the angles of the corresponding complex output elements. If one input is a scalar, it maps to the corresponding component (magnitude or angle) of all the complex output signals.

Data Types: single | double | fixed point

Parameters

Input — Type of input
Magnitude (default) | Angle | Magnitude and angle

Specify the kind of input: a magnitude input, an angle input, or both.

Programmatic Use
Block Parameter: Input
Type: character vector
Values: 'Magnitude' | 'Angle' | 'Magnitude and angle'
Default: 'Magnitude and angle'

Angle — Phase angle of output
θ (default) | real-valued scalar, vector, or matrix

Constant phase angle of the output signal, in rad. To compute the CORDIC approximation, the input angle must be between:

- [-2π, 2π) rad, for signed fixed-point types
- [0, 2π) rad, for unsigned fixed-point types

For more information, see “Effect of Out-of-Range Input on CORDIC Approximations“ on page 1-1219.

Dependencies

To enable this parameter, set Input to Magnitude.

Programmatic Use
Block Parameter: ConstantPart
**Type:** character vector  
**Values:** constant scalar  
**Default:** '0'

**Magnitude — Magnitude of output**  
θ (default) | real-valued scalar, vector, or matrix

Constant magnitude of the output signal, specified as a real-valued scalar, vector, or matrix.

**Dependencies**

To enable this parameter, set **Input** to **Angle**.

**Programmatic Use**  
**Block Parameter:** ConstantPart  
**Type:** character vector  
**Values:** real-valued scalar, vector, or matrix  
**Default:** '0'

**Approximation method — CORDIC or none**  
None (default) | CORDIC

Specify the type of approximation for computing output.

<table>
<thead>
<tr>
<th>Approximation Method</th>
<th>Data Types Supported</th>
<th>When to Use This Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>None (default)</td>
<td>Floating point</td>
<td>You want to use the default Taylor series algorithm.</td>
</tr>
<tr>
<td>CORDIC</td>
<td>Floating point and fixed point</td>
<td>You want a fast, approximate calculation.</td>
</tr>
</tbody>
</table>

When you use the CORDIC approximation, follow these guidelines for the input angle:

- For signed fixed-point types, the input angle must fall within the range [-2π, 2π) rad.
- For unsigned fixed-point types, the input angle must fall within the range [0, 2π) rad.

The block uses the following data type propagation rules:
<table>
<thead>
<tr>
<th>Data Type of Magnitude Input</th>
<th>Approximation Method</th>
<th>Data Type of Complex Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Floating point</td>
<td>None or CORDIC</td>
<td>Same as input</td>
</tr>
<tr>
<td>Signed, fixed point</td>
<td>CORDIC</td>
<td>fixdt(1, WL + 2, FL)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>where WL and FL are the word length and fraction length of the magnitude</td>
</tr>
<tr>
<td>Unsigned, fixed point</td>
<td>CORDIC</td>
<td>fixdt(1, WL + 3, FL)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>where WL and FL are the word length and fraction length of the magnitude</td>
</tr>
</tbody>
</table>

**Programmatic Use**

**Block Parameter:** ApproximationMethod

*Type:* character vector  
*Values:* 'None' | 'CORDIC'  
*Default:* 'None'

**Number of iterations — Number of iterations for CORDIC algorithm**

11 (default) | positive integer, less than or equal to word length of fixed-point input

Number of iterations to perform the CORDIC algorithm. The range of possible values depends on the data type of the input:

<table>
<thead>
<tr>
<th>Data Type of Block Inputs</th>
<th>Value You Can Specify</th>
</tr>
</thead>
<tbody>
<tr>
<td>Floating point</td>
<td>A positive integer</td>
</tr>
<tr>
<td>Fixed point</td>
<td>A positive integer that does not exceed the word length of the magnitude input or the word length of the phase angle input, whichever value is smaller</td>
</tr>
</tbody>
</table>

**Dependencies**

To enable this parameter, set **Approximation method** to CORDIC.

**Programmatic Use**

**Block Parameter:** NumberOfIterations

*Type:* character vector  
*Values:* positive integer, less than or equal to word length of fixed-point input
Default: '11'

Scale output by reciprocal of gain factor — Scale real and imaginary parts of complex output
on (default) | off

Select this check box to scale the real and imaginary parts of the complex output by a factor of \((1/CORDIC \text{ gain})\). This value depends on the number of iterations you specify. As the number of iterations goes up, the value approaches 1.647.

This check box is selected by default, which leads to a more numerically accurate result for the complex output, \(X + iY\). However, scaling the output adds two extra multiplication operations, one for \(X\) and one for \(Y\).

Dependencies

To enable this parameter, set Approximation method to CORDIC.

Programmatic Use

Block Parameter: ScaleReciprocalGainFactor
Type: character vector
Values: 'on' | 'off'
Default: 'on'

Sample time — Specify sample time as a value other than -1
-1 (default) | scalar | vector

Specify the sample time as a value other than -1. For more information, see “Specify Sample Time”.

Dependencies

This parameter is not visible unless it is explicitly set to a value other than -1. To learn more, see “Blocks for Which Sample Time Is Not Recommended”.

Programmatic Use

Block Parameter: SampleTime
Type: character vector
Values: scalar or vector
Default: '-1'
Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>double</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
</tr>
</tbody>
</table>

More About

CORDIC

CORDIC is an acronym for COordinate Rotation DIgital Computer. The Givens rotation-based CORDIC algorithm is one of the most hardware-efficient algorithms available because it requires only iterative shift-add operations (see References). The CORDIC algorithm eliminates the need for explicit multipliers. Using CORDIC, you can calculate various functions, such as sine, cosine, arc sine, arc cosine, arc tangent, and vector magnitude. You can also use this algorithm for divide, square root, hyperbolic, and logarithmic functions.

Increasing the number of CORDIC iterations can produce more accurate results, but doing so also increases the expense of the computation and adds latency.

References


Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture
This block has multi-cycle implementations that introduce additional latency in the generated code. To see the added latency, view the generated model or validation model. See “Generated Model and Validation Model” (HDL Coder).

<table>
<thead>
<tr>
<th>Block configuration with additional latency</th>
<th>Number of additional cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Approximation method is CORDIC</td>
<td>Number of iterations + 1</td>
</tr>
</tbody>
</table>

HDL Block Properties

<table>
<thead>
<tr>
<th>ConstrainedOutput Pipeline</th>
<th>Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. For more details, see “ConstrainedOutputPipeline” (HDL Coder).</th>
</tr>
</thead>
</table>
InputPipeline | Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “InputPipeline” (HDL Coder).

OutputPipeline | Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “OutputPipeline” (HDL Coder).

Restrictions
The Magnitude-Angle to Complex block supports HDL code generation when you set Approximation method to CORDIC.

Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

The Magnitude-Angle to Complex block supports fixed-point and base integer data types when you set Approximation method to CORDIC.

See Also
Complex to Magnitude-Angle | Complex to Real-Imag | Real-Imag to Complex

Topics
“Complex Signals”

Introduced before R2006a
Manual Switch

Switch between two inputs

Library: Simulink / Signal Routing

Description

The Manual Switch block is a toggle switch that selects one of its two inputs to pass through to the output. To toggle between inputs, double-click the block. You control the signal flow by setting the switch before you start the simulation or by changing the switch while the simulation is executing. The Manual Switch block retains its current state when you save the model.

Note Double-clicking the Manual Switch block does not open the block dialog box. Instead, it toggles the input choice.

Ports

Input

Port_1 — First input signal
scalar | vector

First of two inputs to the Manual Switch block. The block propagates the selected input to the output. To select the input signal, toggle the switch by double-clicking the block.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus

Port_2 — Second input signal
scalar | vector
Second of two inputs to the Manual Switch block. The block propagates the selected input
to the output. To select the input signal, toggle the switch by double-clicking the block.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus

**Output**

**Port_1 — Output signal**
scalar | vector

Output signal propagated from either the first or second input signal.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus

**Parameters**

To view the block parameters, right-click the block and select Block Parameters (ManualSwitch).

**Allow the two inputs to differ in size (Results in variable-size output signal) — Allow inputs of different sizes**
off (default) | on

Select this check box to allow inputs with different sizes. If you select the box, the block allows inputs with different sizes, and propagates the selected input signal size to the output signal. If you clear the box, the block expands scalar inputs to have the same dimensions as nonscalar inputs. See “Scalar Expansion of Inputs and Parameters”.

**Programmatic Use**
Parameter: varsizes
Type: character vector
Value: 'on' | 'off'
Default: 'off'

**Sample time — Specify sample time as a value other than -1**
-1 (default) | scalar | vector

Specify the sample time as a value other than -1. For more information, see “Specify Sample Time”.
Dependencies

This parameter is not visible unless it is explicitly set to a value other than -1. To learn more, see “Blocks for Which Sample Time Is Not Recommended”.

Programmatic Use
Block Parameter: SampleTime
Type: character vector
Values: scalar or vector
Default: '-1'

Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>bus</th>
<th>double</th>
<th>enumerated</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
<th>string</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

Not recommended for production code.

Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.
See Also
Manual Variant Sink | Manual Variant Source | Multiport Switch | Switch

Introduced before R2006a
Manual Variant Sink

Switch between multiple variant choices at output

**Library:** Simulink / Signal Routing

---

**Description**

The Manual Variant Sink block is a toggle switch that activates one of its variant choices at the output to pass the input.

The block can have two or more output ports and has one input port. Each output port is associated with a variant control. To change the number of output ports, right-click the block and select **Mask Parameters**, then type a value in the **Number of choices** box.

To toggle between the variant choices at output, double-click the block. The block displays the active choice with a line connecting the input to the output. The block propagates the active variant choice at output and discards the blocks connected to inactive output ports during simulation.

**Note** Double-clicking the Manual Variant Sink block does not open the block dialog box instead it toggles the output choice.

---

**Ports**

**Input**

*Port_1 — Input signal*

Scalar | Vector

Input signal passed to the active output port.
Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus

**Output**

**Port_1 — First variant output**
scalar | vector

First variant output signal. The block passes the input signal to this output port when you connect the toggle switch to this port.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus

**Port_n — nth variant output**
scalar | vector

nth variant output signal. The block passes the input signal to this output port when you connect the toggle switch to this port.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus

**Parameters**

To access the block parameter, right-click the block and select **Mask > Mask Parameters**.

**Number of choices — Number of output choices**
2 (default) | scalar

Specify the number of variant output ports.

**Programmatic Use**

**Block Parameter:** NumChoices

**Type:** character vector

**Value:** integer

**Default:** '2'
## Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>bus</th>
<th>double</th>
<th>enumerated</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
<th>string</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

## Extended Capabilities

### C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

### Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

### See Also
Manual Variant Source | Variant Sink | Variant Source

### Topics
- “Introduction to Variant Controls”
- “Define, Configure, and Activate Variants”
- “Working with Variant Choices”
- “Variant Systems” (Embedded Coder)
- “Represent Variant Source and Sink Blocks in Generated Code” (Embedded Coder)
- “Variants Example Models”
Introduced in R2016b
Manual Variant Source

Switch between multiple variant choices at input
Library: Simulink / Signal Routing

Description

The Manual Variant Source block is a toggle switch that activates one of its variant choices at the input to pass through to the output.

A Manual Variant Source block can have two or more input ports and has one output port. Each input port is associated with a variant control. To change the number of input ports, right-click the block and select Mask Parameters, then type a value in the Number of choices box.

To toggle between the variant choices at input, double-click the block. The block displays the active choice with a line connecting the input to the output. The block propagates the active variant choice at input directly to the output and discards the blocks connected to inactive input ports during simulation.

**Note** Double-clicking the Manual Variant Source block does not open the block dialog box instead it toggles the output choice.

Ports

Input

Port_1 — First variant input signal
scalar | vector
First variant input signal. The block passes this input signal to the output port when you connect the toggle switch to this port.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus

Port_n — nth variant input signal
scalar | vector

nth variant input signal. The block passes this input signal to the output port when you connect the toggle switch to this port.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus

Output

Port_1 — Output signal
scalar | vector

Output signal passed from the active variant input signal.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus

Parameters

To access the block parameter, right-click the block and select Mask > Mask Parameters.

Number of choices — Number of input choices
2 (default) | scalar

Specify the number of variant input ports.

Programmatic Use
Block Parameter: NumChoices
Type: character vector
Value: integer
Default: '2'
Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>bus</th>
<th>double</th>
<th>enumerated</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
<th>string</th>
</tr>
</thead>
</table>

Direct Feedthrough  no

Multidimensional Signals  no

Variable-Size Signals  no

Zero-Crossing Detection  no

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

See Also
Manual Variant Source and Sink Blocks | Variant Sink | Variant Source

Topics
“Introduction to Variant Controls”
“Define, Configure, and Activate Variants”
“Working with Variant Choices”
“Variant Systems” (Embedded Coder)
“Represent Variant Source and Sink Blocks in Generated Code” (Embedded Coder)
“Variants Example Models”
Introduced in R2016b
Math Function

Perform mathematical function

**Library:**
- Simulink / Math Operations
- HDL Coder / Math Operations

---

**Description**

The Math Function block performs numerous common mathematical functions.

**Tip** To perform square root calculations, use the Sqrt block.

You can select one of these functions from the **Function** parameter list.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Mathematical Expression</th>
<th>MATLAB Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>exp</td>
<td>Exponential</td>
<td>$e^u$</td>
<td>exp</td>
</tr>
<tr>
<td>log</td>
<td>Natural logarithm</td>
<td>$\ln u$</td>
<td>log</td>
</tr>
<tr>
<td>$10^u$</td>
<td>Power of base 10</td>
<td>$10^u$</td>
<td>$10.^u$ (see power)</td>
</tr>
<tr>
<td>log10</td>
<td>Common (base 10) logarithm</td>
<td>$\log u$</td>
<td>log10</td>
</tr>
<tr>
<td>magnitude^2</td>
<td>Complex modulus</td>
<td>$</td>
<td>u</td>
</tr>
<tr>
<td>square</td>
<td>Power 2</td>
<td>$u^2$</td>
<td>$u.^2$ (see power)</td>
</tr>
<tr>
<td>pow</td>
<td>Power</td>
<td>$\text{sign}(u)*</td>
<td>u</td>
</tr>
<tr>
<td>conj</td>
<td>Complex conjugate</td>
<td>$\bar{u}$</td>
<td>conj</td>
</tr>
</tbody>
</table>
The block output is the result of the operation of the function on the input or inputs. The functions support these types of operations.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Mathematical Expression</th>
<th>MATLAB Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>reciprocal</td>
<td>Reciprocal</td>
<td>$1/u$</td>
<td>$1./u$ (see <code>rdivide</code>)</td>
</tr>
<tr>
<td>hypot</td>
<td>Square root of sum squares</td>
<td>$(u^2+v^2)^{0.5}$</td>
<td><code>hypot</code></td>
</tr>
<tr>
<td>rem</td>
<td>Remainder after division</td>
<td>—</td>
<td><code>rem</code></td>
</tr>
<tr>
<td>mod</td>
<td>Modulus after division</td>
<td>—</td>
<td><code>mod</code></td>
</tr>
<tr>
<td>transpose</td>
<td>Transpose</td>
<td>$u^T$</td>
<td><code>u.'</code> (see “Array vs. Matrix Operations” (MATLAB))</td>
</tr>
<tr>
<td>hermitian</td>
<td>Complex conjugate transpose</td>
<td>$u^H$</td>
<td><code>u'</code> (see “Array vs. Matrix Operations” (MATLAB))</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Function</th>
<th>Scalar Operations</th>
<th>Element-Wise Vector and Matrix Operations</th>
<th>Vector and Matrix Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>exp</td>
<td>Yes</td>
<td>Yes</td>
<td>—</td>
</tr>
<tr>
<td>log</td>
<td>Yes</td>
<td>Yes</td>
<td>—</td>
</tr>
<tr>
<td>$10^u$</td>
<td>Yes</td>
<td>Yes</td>
<td>—</td>
</tr>
<tr>
<td>log10</td>
<td>Yes</td>
<td>Yes</td>
<td>—</td>
</tr>
<tr>
<td>magnitude^2</td>
<td>Yes</td>
<td>Yes</td>
<td>—</td>
</tr>
<tr>
<td>square</td>
<td>Yes</td>
<td>Yes</td>
<td>—</td>
</tr>
<tr>
<td>pow</td>
<td>Yes</td>
<td>Yes</td>
<td>—</td>
</tr>
<tr>
<td>conj</td>
<td>Yes</td>
<td>Yes</td>
<td>—</td>
</tr>
<tr>
<td>reciprocal</td>
<td>Yes</td>
<td>Yes</td>
<td>—</td>
</tr>
</tbody>
</table>
### Scalar Operations

<table>
<thead>
<tr>
<th>Function</th>
<th>Scalar Operations</th>
<th>Element-Wise Vector and Matrix Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>hypot</td>
<td>Yes, on two inputs</td>
<td>Yes, on two inputs (two vectors or two matrices of the same size, a scalar and a vector, or a scalar and a matrix)</td>
</tr>
<tr>
<td>rem</td>
<td>Yes, on two inputs</td>
<td>Yes, on two inputs (two vectors or two matrices of the same size, a scalar and a vector, or a scalar and a matrix)</td>
</tr>
<tr>
<td>mod</td>
<td>Yes, on two inputs</td>
<td>Yes, on two inputs (two vectors or two matrices of the same size, a scalar and a vector, or a scalar and a matrix)</td>
</tr>
</tbody>
</table>

### Element-Wise Vector and Matrix Operations

<table>
<thead>
<tr>
<th>Function</th>
<th>Element-Wise Vector and Matrix Operations</th>
<th>Vector and Matrix Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>transpose</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td>hermitian</td>
<td>—</td>
<td>Yes</td>
</tr>
</tbody>
</table>

The name of the function appears on the block. The appropriate number of input ports appears automatically.

**Tip** Use the Math Function block when you want vector or matrix output.

## Data Type Support

This table shows the input data types that each function of the block can support.

<table>
<thead>
<tr>
<th>Function</th>
<th>single</th>
<th>double</th>
<th>boolean</th>
<th>built-in integer</th>
<th>fixed point</th>
</tr>
</thead>
<tbody>
<tr>
<td>exp</td>
<td>Yes</td>
<td>Yes</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>log</td>
<td>Yes</td>
<td>Yes</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>10^u</td>
<td>Yes</td>
<td>Yes</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>
### Ports

#### Input

**Port_1 — Input signal**

scalar | vector | matrix

Input signal specified as a scalar, vector, or matrix. All supported modes accept both real and complex inputs, except for reciprocal, which does not accept complex fixed-point inputs. See Description on page 1-1241 for more information.

**Dependencies**

Data type support for this block depends on the **Function** you select and the size of the input(s). For more information, see “Data Type Support” on page 1-1243.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point

**Port_2 — Input signal**

scalar | vector | matrix
Input signal specified as a scalar, vector, or matrix. All supported modes accept both real and complex inputs, except for reciprocal, which does not accept complex fixed-point inputs.

**Dependencies**

To enable this port, set **Function** to hypot, rem, or mod.

Data type support for this block depends on the **Function** you select, and the size of the input(s). For more information, see “Data Type Support” on page 1-1243.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point

**Output**

**Port_1 — Result of the operation of the function on the input or inputs**

scalar | vector | matrix

Output signal specified as a scalar, vector, or matrix. The dimensions of the block output depend on the **Function** you select and the size of the inputs. The block output is real or complex, depending on what you select for **Output signal type**. See Description on page 1-1241 for more information.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point

**Parameters**

**Main**

**Function — Math function**

exp (default) | log | 10^u | log10 | magnitude^2 | square | pow | conj | reciprocal | hypot | rem | mod | transpose | hermitian

Specify the mathematical function. See Description on page 1-1241 for more information about the options for this parameter.

**Dependency**

Setting **Function** to pow enables the **Signed power** parameter.
Programmatic Use

Block Parameter: Operator

Type: character vector

Values: 'exp' | 'log' | '10^u' | 'log10' | 'magnitude^2' | 'square' | 'pow' | 'conj' | 'reciprocal' | 'hypot' | 'rem' | 'mod' | 'transpose' | 'hermitian'

Default: 'exp'

Signed power — Power signedness

on (default) | off

Take into account sign of the input signal when calculating power, specified as on or off.

• on — Calculate power of the absolute value of the input, multiplied by the sign of the input.
• off — Calculate power of the actual input value. If the first input is negative and the second input is a fractional exponent, return nan.

Dependency

Setting Function to pow enables this parameter.

Programmatic Use

Block Parameter: SignedPower

Type: character vector

Values: 'on' | 'off'

Default: 'on'

Output signal type — Complexity of output signal

auto (default) | real | complex

Specify the output signal type of the Math Function block as auto, real, or complex.

<table>
<thead>
<tr>
<th>Function</th>
<th>Input Signal Type</th>
<th>Output Signal Type</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Auto</td>
</tr>
<tr>
<td>exp, log, log10,</td>
<td>real</td>
<td>real</td>
</tr>
<tr>
<td>10^u, square,</td>
<td>complex</td>
<td></td>
</tr>
<tr>
<td>pow, reciprocal,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>conjugate,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>transpose,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>hermitian</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Function</td>
<td>Input Signal Type</td>
<td>Output Signal Type</td>
</tr>
<tr>
<td>----------------</td>
<td>-------------------</td>
<td>--------------------</td>
</tr>
<tr>
<td>magnitude squared</td>
<td>real</td>
<td>real</td>
</tr>
<tr>
<td></td>
<td>complex</td>
<td>real</td>
</tr>
<tr>
<td></td>
<td></td>
<td>complex</td>
</tr>
<tr>
<td>hypot, rem, mod</td>
<td>real</td>
<td>real</td>
</tr>
<tr>
<td></td>
<td>complex</td>
<td>real</td>
</tr>
<tr>
<td></td>
<td></td>
<td>complex</td>
</tr>
</tbody>
</table>

**Programmatic Use**

**Block Parameter:** OutputSignalType  
**Type:** character vector  
**Values:** 'auto' | 'real' | 'complex'  
**Default:** 'auto'

**Sample time — Specify sample time as a value other than -1**  
-1 (default) | scalar | vector

Specify the sample time as a value other than -1. For more information, see “Specify Sample Time”.

**Dependencies**

This parameter is not visible unless it is explicitly set to a value other than -1. To learn more, see “Blocks for Which Sample Time Is Not Recommended”.

**Programmatic Use**

**Block Parameter:** SampleTime  
**Type:** character vector  
**Values:** scalar or vector  
**Default:** '-1'

**Signal Attributes**

**Output minimum — Minimum output value for range checking**  
[] (default) | scalar

Lower value of the output range that Simulink checks.

Simulink uses the minimum to perform:
• Parameter range checking (see “Specify Minimum and Maximum Values for Block
Parameters”) for some blocks.
• Simulation range checking (see “Signal Ranges” and “Enable Simulation Range
Checking”).
• Automatic scaling of fixed-point data types.
• Optimization of the code that you generate from the model. This optimization can
remove algorithmic code and affect the results of some simulation modes such as SIL
or external mode. For more information, see “Optimize using the specified minimum
and maximum values” (Simulink Coder).

Note Output minimum does not saturate or clip the actual output signal. Use the
Saturation block instead.

Programmatic Use
Block Parameter: OutMin
Type: character vector
Values: ' [ ] ' | scalar
Default: ' [ ] '

Output maximum — Maximum output value for range checking
[ ] (default) | scalar

Upper value of the output range that Simulink checks.

Simulink uses the maximum value to perform:
• Parameter range checking (see “Specify Minimum and Maximum Values for Block
Parameters”) for some blocks.
• Simulation range checking (see “Signal Ranges” and “Enable Simulation Range
Checking”).
• Automatic scaling of fixed-point data types.
• Optimization of the code that you generate from the model. This optimization can
remove algorithmic code and affect the results of some simulation modes such as SIL
or external mode. For more information, see “Optimize using the specified minimum
and maximum values” (Simulink Coder).
**Note** Output maximum does not saturate or clip the actual output signal. Use the Saturation block instead.

**Programmatic Use**

**Block Parameter:** OutMax

**Type:** character vector

**Values:** '[]' | scalar

**Default:** '[]'

**Output data type — Specify the output data type**

Inherit: Same as first input (default) | Inherit: Inherit via internal rule | Inherit: Inherit via back propagation | double | single | int8 | uint8 | int16 | uint16 | int32 | uint32 | int64 | uint64 | fixdt(1,16) | fixdt(1,16,0) | fixdt(1,16,2^0,0) | <data type expression>

Specify the output data type. You can set it to:

- A rule that inherits a data type, for example, Inherit: Inherit via back propagation
- The name of a built-in data type, for example, single
- The name of a data type object, for example, a Simulink.NumericType object
- An expression that evaluates to a data type, for example, fixdt(1,16,0)

Click the **Show data type assistant** button to display the **Data Type Assistant**, which helps you set the data type attributes. For more information, see “Specify Data Types Using Data Type Assistant”.

**Dependencies**

To enable this parameter, set the **Function** to magnitude^2, square, or reciprocal.

**Programmatic Use**

**Block Parameter:** OutDataTypeStr

**Type:** character vector

**Values:** 'Inherit: Inherit via internal rule' | 'Inherit: Same as first input' | 'Inherit: Inherit via back propagation' | 'double' | 'single' | 'int8' | 'uint8' | 'int16' | 'uint16' | 'int32' | 'uint32' | 'int64' | 'uint64' | 'fixdt(1,16)' | 'fixdt(1,16,0)' | 'fixdt(1,16,2^0,0)' | '<data type expression>'

1-1249
Default: 'Inherit: Same as first input'

Lock output data type setting against changes by the fixed-point tools — Prevent fixed-point tools from overriding data types
off (default) | on

Select this parameter to prevent the fixed-point tools from overriding the Output data type you specify on the block. For more information, see “Use Lock Output Data Type Setting” (Fixed-Point Designer).

Dependencies

To enable this parameter, set the Function to magnitude^2, square, or reciprocal.

Programmatic Use

Block Parameter: LockScale
Type: character vector
Values: 'off' | 'on'
Default: 'off'

Integer rounding mode — Rounding mode for fixed-point operations
Floor (default) | Ceiling | Convergent | Nearest | Round | Simplest | Zero

Rounding mode for fixed-point operations. For more information, see “Rounding” (Fixed-Point Designer).

Block parameters always round to the nearest representable value. To control the rounding of a block parameter, enter an expression using a MATLAB rounding function into the mask field.

Dependencies

To enable this parameter, set the Function to magnitude^2, square, or reciprocal.

Programmatic Use

Block Parameter: RndMeth
Type: character vector
Values: 'Ceiling' | 'Convergent' | 'Floor' | 'Nearest' | 'Round' | 'Simplest' | 'Zero'
Default: 'Floor'

Saturate on integer overflow — Choose the behavior when integer overflow occurs
on (default) | boolean
<table>
<thead>
<tr>
<th>Action</th>
<th>Reasons for Taking This Action</th>
<th>What Happens for Overflows</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Select this check box.</td>
<td>Your model has possible overflow, and you want explicit saturation protection in the generated code.</td>
<td>Overflows saturate to either the minimum or maximum value that the data type can represent.</td>
<td>The maximum value that the int8 (signed, 8-bit integer) data type can represent is 127. Any block operation result greater than this maximum value causes overflow of the 8-bit integer. With the check box selected, the block output saturates at 127. Similarly, the block output saturates at a minimum output value of -128.</td>
</tr>
<tr>
<td>Do not select this check box.</td>
<td>You want to optimize efficiency of your generated code. You want to avoid overspecifying how a block handles out-of-range signals. For more information, see “Check for Signal Range Errors”.</td>
<td>Overflows wrap to the appropriate value that is representable by the data type.</td>
<td>The maximum value that the int8 (signed, 8-bit integer) data type can represent is 127. Any block operation result greater than this maximum value causes overflow of the 8-bit integer. With the check box cleared, the software interprets the overflow-causing value as int8, which can produce an unintended result. For example, a block result of 130 (binary 1000 0010) expressed as int8, is -126.</td>
</tr>
</tbody>
</table>

When you select this check box, saturation applies to every internal operation on the block, not just the output or result. Usually, the code generation process can detect when overflow is not possible. In this case, the code generator does not produce saturation code.
Dependencies

To enable this parameter, set the Function to magnitude^2, square, conj, reciprocal, or hermitian.

Programmatic Use

**Block Parameter:** SaturateOnIntegerOverflow  
**Type:** character vector  
**Value:** 'off' | 'on'  
**Default:** 'on'

Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>double</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Extended Capabilities

**C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

**HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.
**HDL Architecture**

You can generate HDL code for the Math architecture in native floating-point mode.

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ComplexConjugate</td>
<td>Compute complex conjugate.</td>
</tr>
<tr>
<td>Hermitian</td>
<td>Compute hermitian.</td>
</tr>
<tr>
<td>Transpose</td>
<td>Compute array transpose. See Math Function in the Simulink documentation.</td>
</tr>
</tbody>
</table>

**Reciprocal Architecture**

This block has multi-cycle implementations that introduce additional latency in the generated code. To see the added latency, view the generated model or validation model. See “Generated Model and Validation Model” (HDL Coder).

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Parameters</th>
<th>Additional Cycles of Latency</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reciprocal</td>
<td>None</td>
<td>0</td>
<td>Compute reciprocal as $1/N$, using the HDL divide (/) operator to implement the division.</td>
</tr>
</tbody>
</table>
| ReciprocalRsqrtBase | Iterations   | Signed input: $\text{Iterations} + 5$  
Unsigned input: $\text{Iterations} + 3$ | Use the iterative Newton method. Select this option to optimize area.  
The default value for $\text{Iterations}$ is 3.  
The recommended value for $\text{Iterations}$ is from 2 through 10. If $\text{Iterations}$ is outside the recommended range, HDL Coder generates a message. |
### Architecture Parameters Additional Cycles of Latency Description

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ReciprocalRsqrtBasedNewtonSingleRate</td>
<td>Iterations</td>
<td>Signed input: ((\text{Iterations} \times 4) + 8)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Unsigned input: ((\text{Iterations} \times 4) + 6)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Use the single rate pipelined Newton method. Select this option to optimize speed, or if you want a single rate implementation. The default value for Iterations is 3. The recommended value for Iterations is from 2 through 10. If Iterations is outside the recommended range, the coder generates a message.</td>
</tr>
</tbody>
</table>

The Newton-Raphson iterative method:

\[
x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} = x_i(1.5 - 0.5a x_i^2)
\]

ReciprocalRsqrtBasedNewton and ReciprocalRsqrtBasedNewtonSingleRate implement the Newton-Raphson method with:

\[
f(x) = \frac{1}{x^2} - 1
\]

### HDL Block Properties

<table>
<thead>
<tr>
<th>General</th>
<th>Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. For more details, see “ConstrainedOutputPipeline” (HDL Coder).</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConstranedOutputPipeline</td>
<td>Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. For more details, see “ConstrainedOutputPipeline” (HDL Coder).</td>
</tr>
</tbody>
</table>
### General

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>InputPipeline</strong></td>
<td>Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “InputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td><strong>OutputPipeline</strong></td>
<td>Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “OutputPipeline” (HDL Coder).</td>
</tr>
</tbody>
</table>

### Native Floating Point

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CheckResetToZero</strong></td>
<td>Use this property for the mod and rem functions of the Math Function block. If you have numbers a and b such that the quotient a/b is close to an integer, this setting treats a as an integral multiple of b, and rem(a,b) = 0. This result is numerically accurate and matches the simulation results. However, computing this result uses additional resources and increases the area footprint on the target FPGA device. For more information, see “CheckResetToZero” (HDL Coder).</td>
</tr>
<tr>
<td><strong>HandleDenormals</strong></td>
<td>Specify whether you want HDL Coder to insert additional logic to handle denormal numbers in your design. Denormal numbers are numbers that have magnitudes less than the smallest floating-point number that can be represented without leading zeros in the mantissa. The default is inherit. See also “HandleDenormals” (HDL Coder).</td>
</tr>
<tr>
<td><strong>LatencyStrategy</strong></td>
<td>Specify whether to map the blocks in your design to inherit, Max, Min, Zero, or Custom for the floating-point operator. The default is inherit. See also “LatencyStrategy” (HDL Coder).</td>
</tr>
<tr>
<td><strong>NFPCustomLatency</strong></td>
<td>To specify a value, set LatencyStrategy to Custom. HDL Coder adds latency equal to the value that you specify for the NFPCustomLatency setting. See also “NFPCustomLatency” (HDL Coder).</td>
</tr>
</tbody>
</table>
Native Floating Point

| MaxIterations | Use this property for the mod and rem functions of the Math Function block. If you have numbers \(a\) and \(b\) that are significantly large integers, you can increase MaxIterations to match the simulation results. However, computing this result uses additional resources and increases the area footprint on the target FPGA device. For more information, see “MaxIterations” (HDL Coder). |

Complex Data Support

The \(\text{conj}\), \(\text{hermitian}\), and \(\text{transpose}\) functions support complex data.

Restrictions

When you use a reciprocal implementation:

- Input must be scalar and must have integer or fixed-point (signed or unsigned) data type.
- The output must be scalar and have integer or fixed-point (signed or unsigned) data type.
- Only the Zero rounding mode is supported.
- The Saturate on integer overflow option on the block must be selected.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

Design and simulate fixed-point systems using Fixed-Point Designer™.

The Math Function block only supports fixed-point conversion in certain configurations. For more information, see the Block Support Table.

See Also

Sqrt, Signed Sqrt, Reciprocal Sqrt | Trigonometric Function
**Topics**

“hisl_0004: Usage of Math Function blocks (natural logarithm and base 10 logarithm)"

**Introduced before R2006a**
MATLAB Function

Include MATLAB code in models that generate embeddable C code

**Library:**  
Simulink / User-Defined Functions  
HDL Coder / User-Defined Functions

**Description**

With a MATLAB Function block, you can write a MATLAB function for use in a Simulink model. The MATLAB function you create executes for simulation and generates code for a Simulink Coder target. If you are new to the Simulink and MATLAB products, see “Implementing MATLAB Functions Using Blocks” and “Create Custom Functionality Using MATLAB Function Block” for an overview.

Double-clicking the MATLAB Function block opens its editor, where you write the MATLAB function, as in this example:

```matlab
function [mean, stdev] = stats(vals)
  % #codegen

  % calculates a statistical mean and a standard deviation for the values in vals.
  len = length(vals);
  mean = avg(vals, len);
  stdev = sqrt(sum(((vals-avg(vals, len)).^2))/len);
  coder.extrinsic('plot');
  plot(vals,'-+');

  function mean = avg(array, size)
  mean = sum(array)/size;
```
To learn more about this editor, see “MATLAB Function Block Editor”.

You specify input and output data to the MATLAB Function block in the function header as arguments and return values. The argument and return values of the preceding example function correspond to the inputs and outputs of the block in the model:

![MATLAB Function block example]

You can also define data, input triggers, and function call outputs using the Ports and Data Manager, which you access from the MATLAB Function Block Editor by selecting Edit Data. See “Ports and Data Manager”.

The MATLAB Function block generates efficient embeddable code based on an analysis that determines the size, class, and complexity of each variable. This analysis imposes the following restrictions:

- The first assignment to a variable defines its size, class, and complexity.
  
- You cannot reassign variable properties after the initial assignment except when using variable-size data or reusing variables in the code for different purposes.

  See “Reassignment of Variable Properties”.

In addition to language restrictions, the MATLAB Function block supports a subset of the functions available in MATLAB. A list of supported functions is given in “Functions and Objects Supported for C/C++ Code Generation”. These functions include functions in common categories, such as:

- Arithmetic operators like plus, minus, and power. For more information, see “Array vs. Matrix Operations” (MATLAB).
- Matrix operations like size, and length
• Advanced matrix operations like lu, inv, svd, and chol
• Trigonometric functions like sin, cos, sinh, and cosh

**Note** Although the code for this block attempts to produce exactly the same results as MATLAB, differences might occur due to rounding errors. These numerical differences, which might be a few eps initially, can magnify after repeated operations. Reliance on the behavior of nan is not recommended. Different C compilers can yield different results for the same computation.

**Note** In the MATLAB Function block, the %#codegen directive is included to emphasize that the block’s MATLAB algorithm is always intended for code generation. The %#codegen directive, or the absence of it, does not change the error checking behavior in the context of the MATLAB Function block. For more information see “Compilation Directive %#codegen”.

To support visualization of data, the MATLAB Function block supports calls to MATLAB functions for simulation only. See “Extrinsic Functions” to understand some of the limitations of this capability, and how it integrates with code analysis for this block. If these function calls do not directly affect any of the Simulink inputs or outputs, the calls do not appear in Simulink Coder generated code.

From MATLAB Function blocks, you can also call functions defined in a Simulink Function block. You can call Stateflow functions with **Export Chart Level Functions (Make Global)** and **Allow exported functions to be called by Simulink** checked in the chart Properties dialog box.

In the Ports and Data Manager, you can declare a block input to be a Simulink parameter instead of a port. The MATLAB Function block also supports inheritance of types and size for inputs, outputs, and parameters. You can also specify these properties explicitly. See “Type Function Arguments“, “Size Function Arguments“, and “Add Parameter Arguments” for descriptions of variables that you use in MATLAB Function blocks.

Recursive calls are not allowed in MATLAB Function blocks.

By default, MATLAB Function blocks have direct feedthrough enabled. To disable it, in the Ports and Data Manager, clear the **Allow direct feedthrough** check box. Nondirect feedthrough enables semantics to ensure that outputs rely only on current state. Using nondirect feedthrough enables you to use MATLAB Function blocks in a feedback loop and prevent algebraic loops.
Ports

Input

**u — Input argument u**

scalar | vector | matrix

Input corresponding to the first input argument of the function inside the MATLAB Function block. If you rename the function argument in the editor, the block renames the port correspondingly.

Data types supported by MATLAB but not supported by Simulink may not be passed between the Simulink model and the function within the MATLAB Function block. These types may be used within the MATLAB Function block.

For more information on fixed-point support for this block, refer to “Fixed-Point Data Types with MATLAB Function Block” (Fixed-Point Designer) and “MATLAB Function Block with Data Type Override” (Fixed-Point Designer).

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus

**n — Input argument n**

scalar | vector | matrix

n-th input argument to the function in the MATLAB Function block. When you add the argument in the editor, the block adds the port correspondingly.

Data types supported by MATLAB but not supported by Simulink may not be passed between the Simulink model and the function within the MATLAB Function block. These types may be used within the MATLAB Function block.

For more information on fixed-point support for this block, refer to “Fixed-Point Data Types with MATLAB Function Block” (Fixed-Point Designer) and “MATLAB Function Block with Data Type Override” (Fixed-Point Designer).

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus
**Output**

**y — Output argument**

scalar | vector | matrix

First output argument of the function inside the MATLAB Function block. If you rename the function argument in the editor, the block renames the port correspondingly.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus

**n — Output argument n**

scalar | vector | matrix

nth output argument from the function inside the MATLAB Function block. When you add the argument in the editor, the block adds the port correspondingly.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus

**Block Characteristics**

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>bus</th>
<th>double</th>
<th>enumerated</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>limited</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>limited</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

a. Actual data type or capability support depends on block implementation.
Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

Actual data type or capability support depends on block implementation.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic. Actual data type or capability support depends on block implementation. For best practices of using the MATLAB Function block, see:

- “Design Guidelines for the MATLAB Function Block” (HDL Coder)
- “Generate Instantiable Code for Functions” (HDL Coder)
- “Optimize MATLAB Loops” (HDL Coder)
- “Pipeline MATLAB Expressions” (HDL Coder)

HDL Architecture

<table>
<thead>
<tr>
<th>MATLAB Function (Default)</th>
<th>This is the default architecture for MATLAB Function blocks with fixed-point data types. When you use this architecture, you can use speed and area optimizations for blocks that surround the MATLAB Function block. The code inside the MATLAB Function block is not optimized.</th>
</tr>
</thead>
</table>
MATLAB Datapath | This is the default architecture for MATLAB Function blocks with floating-point data types. When you use this architecture, the code generator treats the MATLAB Function block like a regular Subsystem block. By enabling this architecture for fixed-point types, you can use speed and area optimizations for blocks inside the MATLAB Function block and across the MATLAB Function block with other Simulink blocks. This capability enables you to specify additional properties for the MATLAB Function block that you would specify for a Subsystem such as ClockRatePipelining and FlattenHierarchy.

### HDL Block Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AdaptivePipelining</td>
<td>Automatic pipeline insertion based on the synthesis tool, target frequency, and multiplier word-lengths. The default is inherit. To specify this property, set Architecture to MATLAB Datapath. See also “AdaptivePipelining” (HDL Coder).</td>
</tr>
<tr>
<td>BalanceDelays</td>
<td>Detects introduction of new delays along one path and inserts matching delays on the other paths. The default is inherit. To specify this property, set Architecture to MATLAB Datapath. See also “BalanceDelays” (HDL Coder).</td>
</tr>
<tr>
<td>ClockRatePipelining</td>
<td>Insert pipeline registers at a faster clock rate instead of the slower data rate. The default is inherit. To specify this property, set Architecture to MATLAB Datapath. See also “ClockRatePipelining” (HDL Coder).</td>
</tr>
<tr>
<td>ConstMultiplierOptimization</td>
<td>Canonical signed digit (CSD) or factored CSD optimization. The default is none. See also “ConstMultiplierOptimization” (HDL Coder).</td>
</tr>
<tr>
<td>ConstrainedOutputPipeline</td>
<td>Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. For more details, see “ConstrainedOutputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td>DistributedPipelining</td>
<td>Pipeline register distribution, or register retiming. The default is off. See also “DistributedPipelining” (HDL Coder).</td>
</tr>
<tr>
<td>Property</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>DSPStyle</td>
<td>Synthesis attributes for multiplier mapping. The default is none. To specify this property, set <strong>Architecture</strong> to MATLAB Datapath. See also “DSPStyle” (HDL Coder).</td>
</tr>
<tr>
<td>FlattenHierarchy</td>
<td>Remove subsystem hierarchy from generated HDL code. The default is inherit. To specify this property, set <strong>Architecture</strong> to MATLAB Datapath. See also “FlattenHierarchy” (HDL Coder).</td>
</tr>
<tr>
<td>InputPipeline</td>
<td>Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “InputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td>InstantiateFunctions</td>
<td>Generate a VHDL entity or Verilog module for each function. The default is off. See also “InstantiateFunctions” (HDL Coder).</td>
</tr>
<tr>
<td>LoopOptimization</td>
<td>Unroll, stream, or do not optimize loops. The default is none. If you set <strong>Architecture</strong> to MATLAB Datapath, you can only unroll loops. If you want to use loop streaming with the MATLAB Datapath architecture, you can use the streaming optimization by specifying a <strong>StreamingFactor</strong>. See also “LoopOptimization” (HDL Coder).</td>
</tr>
<tr>
<td>MapPersistentVarsToRAM</td>
<td>Map persistent arrays to RAM. The default is off. See also “MapPersistentVarsToRAM” (HDL Coder).</td>
</tr>
<tr>
<td>OutputPipeline</td>
<td>Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “OutputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td>ResetType</td>
<td>Suppress reset logic generation. The default is default, which generates reset logic. See also “ResetType” (HDL Coder).</td>
</tr>
<tr>
<td>SharingFactor</td>
<td>Number of functionally equivalent resources to map to a single shared resource. The default is 0. See also “Resource Sharing” (HDL Coder).</td>
</tr>
<tr>
<td>StreamingFactor</td>
<td>Number of parallel data paths, or vectors, that are time multiplexed to transform into serial, scalar data paths. The default is 0, which implements fully parallel data paths. To specify this property, set <strong>Architecture</strong> to MATLAB Datapath. See also “Streaming” (HDL Coder).</td>
</tr>
</tbody>
</table>
UseMatrixTypesInHDL

Generate 2-D matrices in HDL code. The default is off. To specify this property, set Architecture to MATLAB Function. See also “UseMatrixTypesInHDL” (HDL Coder).

VariablesToPipeline

Warning VariablesToPipeline is not recommended. Use coder.hdl.pipeline instead.

Insert a pipeline register at the output of the specified variable or variables. Specify the list of variables as a character vector, with spaces separating the variables. To specify this property, set Architecture to MATLAB Function.

Complex Data Support

This block supports code generation for complex signals.

See also “Complex Data Type Support” (HDL Coder).

Tunable Parameter Support

HDL Coder supports both tunable and non-tunable parameters with the following data types:

- Scalar
- Vector
- Complex
- Structure
- Enumeration

For details, see “Generate DUT Ports for Tunable Parameters” (HDL Coder) and .“Design Guidelines for the MATLAB Function Block” (HDL Coder)

Restrictions

- If the block contains a System object™, block inputs cannot have non-discrete (constant or Inf) sample time.
- HDL Coder does not support a MATLAB Function that contains the same variable as the input and output of the function. For example, this MATLAB code is not supported.

```
function y = myFun(y)
    %#codegen
```
\[ y = 3 \times y; \]

For the MATLAB language subset supported for HDL code generation from a MATLAB Function block, see “Supported MATLAB Data Types, Operators, and Control Flow Statements” (HDL Coder).

**PLC Code Generation**
Generate Structured Text code using Simulink® PLC Coder™.

**Fixed-Point Conversion**
Design and simulate fixed-point systems using Fixed-Point Designer™.

Actual data type or capability support depends on block implementation.

**See Also**
Subsystem, Atomic Subsystem, Nonvirtual Subsystem, CodeReuse Subsystem

**Topics**
“Create Custom Functionality Using MATLAB Function Block”
“Use Nondirect Feedthrough in a MATLAB Function Block”
“Implementing MATLAB Functions Using Blocks”
“MATLAB Function Block Capabilities”

**Introduced in R2011a**
MATLAB System

Include System object in model

Library: Simulink / User-Defined Functions
HDL Coder / User-Defined Functions

Description

The MATLAB System block brings existing System objects (based on matlab.System) into Simulink. It also enables you to use System object APIs to develop new blocks for Simulink. For more information on this block, see “MATLAB System Block”.

For interpreted execution, the model simulates the block using the MATLAB execution engine.

For code generation, the model simulates the block using code generation (using the subset of MATLAB code supported for code generation). The MATLAB System block supports only a subset of the functions available in MATLAB. See “Functions and Objects Supported for C/C++ Code Generation” for a complete list of functions. These functions include those in common categories, such as:

- “Array vs. Matrix Operations” (MATLAB), like plus, minus, and power
- Matrix operations, like size and length
- Advanced matrix operations, like lu, inv, svd, and chol
- Trigonometric functions, like sin, cos, sinh, and cosh

System Objects

To use the MATLAB System block, you must first have a new System object or use an existing one. For more information, see “Integrate System Objects Using MATLAB System Block”.

1-1268
Ports

Input

**In — Signal input to a MATLAB System block**

scalar | vector | matrix

The MATLAB System block accepts inputs of most types that Simulink supports. It does not support virtual buses as input or output. It does not support nonvirtual buses that contain variable-size signals. For more information, see “Data Types Supported by Simulink”.

For information on fixed-point support for this block, see “Code Acceleration and Code Generation from MATLAB” (Fixed-Point Designer).

The MATLAB System block supports Simulink frames. For more information, see “Sample-and Frame-Based Concepts” (DSP System Toolbox).

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

Output

**Out — Signal output of a MATLAB System block**

scalar | vector | matrix

Signal output of a MATLAB System block that the System object returns.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus

Parameters

**System Object Name — Name of the System object**

cell array (default)

Specify the full name of the user-defined System object class without the file extension. This entry is case sensitive. The class name must exist on the MATLAB path.

You can specify a System object name in one of these ways:
• Enter the name in the text box.
• Click the list arrow attached to the text box. If valid System objects exist in the current folder, the names appear in the list. Select a System object from this list.
• Browse to a folder that contains a valid System object. If the folder is not on your MATLAB path, the software prompts you to add it.

If you need to create a System object, you can create one from a template by clicking New.

After you save the System object, the name appears in the System object name text box.

Use the full name of the user-defined System object class name. The block does not accept a MATLAB variable that you have assigned to a System object class name.

**Programmatic Use**

**Block Parameter:** System  
**Type:** character vector  
**Value:** name of the System object  
**Default:** ''

**New — Create a System object from a template**

Basic (default) | Advanced | Simulink Extension

Select one of the options for a System object template.

**Basic**

Starts MATLAB Editor and displays a template for a simple System object using the fewest System object methods.

**Advanced**

Starts MATLAB Editor and displays a template for a more advanced System object using most of the System object methods.

**Simulink Extension**

Starts MATLAB Editor and displays a file that contains utilities for customizing the block for Simulink. This is the same file available in MATLAB when you select New > System Object > Simulink Extension.

After you save the System object, you can enter the name in the System object name text box.
Simulate using — Select the simulation mode

Code generation (default) | Interpreted Execution

Select the simulation mode.

Code generation

On the first model run, simulate and generate code for MATLAB System block using only MATLAB functions supported for code generation. If the structure of the block does not change, subsequent model runs do not regenerate the code.

If the simulation mode is Code generation, System Objects accept a maximum of 32 inputs.

Interpreted execution

Simulate model using all supported MATLAB functions. Choosing this option can slow simulation performance.

Dependency — Dependency parameter for MATLAB System block

auto (default)

After you assign a valid System object class name to the block, the next time you open the block dialog box, the parameter is visible. This parameter appears for every MATLAB System block. You cannot remove it.

- If the block has no tabs, this parameter appears at the bottom of the dialog box.
- If the block has multiple tabs, this parameter appears at the bottom of the first tab of the dialog box.

Saturate on integer overflow — Specify whether overflows saturate

Off (default) | On

On

Overflows saturate to either the minimum or maximum value that the data type can represent. For example, an overflow associated with a signed 8-bit integer can saturate to -128 or 127.

Off

Overflows wrap to the appropriate value that the data type can represent. For example, the number 130 does not fit in a signed 8-bit integer and wraps to -126.
**Tips**

- Consider selecting this check box when your model has a possible overflow and you want explicit saturation protection in the generated code.
- Consider clearing this check box when you want to optimize efficiency of your generated code. Clearing this check box also helps you to avoid overspecifying how a block handles out-of-range signals. For more information, see “Check for Signal Range Errors”.
- When you select this check box, saturation applies to every internal operation on the block, not just the output or result.
- In general, the code generation process can detect when overflow is not possible. In this case, the code generator does not produce saturation code.

**Dependency**

This check box appears when you use the showFiSettingsImpl method in the System object.

**Programmatic Use**

**Block Parameter:** SaturateOnIntegerOverflow  
**Type:** character vector  
**Values:** 'off' | 'on'  
**Default:** 'off'

**Treat these inherited Simulink signal types as fi objects — Specify fi data types**

**Fixed-point (default) | Fixed-point & Integer**

Select which inherited data types to treat fi data types,

**Fixed-point**
- Treat fixed-point data types as fi data types.

**Fixed-point & Integer**
- Treat fixed-point and integer data types as fi data types.

**Dependency**

This check box appears when you use the showFiSettingsImpl method in the System object.
MATLAB System fimath — Specify fixed-point settings to use
Same as MATLAB (default) | Specify Other

Select which fixed-point math settings to use.

**Same as MATLAB**

Use the current MATLAB fixed-point math settings.

**Specify Other**

Enable the edit box for specifying the desired fixed-point math settings. For information on setting fixed-point math, see fimath.

**Dependency**

This check box appears when you use the `showFiSettingsImpl` method in the System object.

**Block Characteristics**

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>bus</th>
<th>double</th>
<th>enumerated</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>limited</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>limited</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

a. Actual data type or capability support depends on block implementation.
b. See Nonvirtual Buses and MATLAB System Block for more information.
c. See Variable-Size Signals for more information.
Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

Actual code generation support depends on block implementation.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

You can define a System object and use it in a MATLAB System block for HDL code generation.

Tunable Parameter Support
HDL Coder supports tunable parameters with the following data types:

- Numeric
- Fixed point
- Character
- Logical

When using tunable parameters with the MATLAB System block, the tunable parameter should be a Simulink.Parameter object with the StorageClass set to ExportedGlobal.

```matlab
x = Simulink.Parameter
x.Value = 1
x.CoderInfo.StorageClass = 'ExportedGlobal'
```

For details, see “Generate DUT Ports for Tunable Parameters” (HDL Coder).

HDL Architecture
This block has a single, default HDL architecture.
**HDL Block Properties**

If you use a predefined System object, the HDL block properties available are the same as the properties available for the corresponding block.

By default, the following HDL block properties are available.

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ConstMultiplierOptimization</strong></td>
<td>Canonical signed digit (CSD) or factored CSD optimization. The default is <em>none</em>. See also “ConstMultiplierOptimization” (HDL Coder).</td>
</tr>
<tr>
<td><strong>ConstrainedOutputPipeline</strong></td>
<td>Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. For more details, see “ConstrainedOutputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td><strong>DistributedPipelining</strong></td>
<td>Pipeline register distribution, or register retiming. The default is <em>off</em>. See also “DistributedPipelining” (HDL Coder).</td>
</tr>
<tr>
<td><strong>InputPipeline</strong></td>
<td>Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “InputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td><strong>LoopOptimization</strong></td>
<td>Unroll, stream, or do not optimize loops. The default is <em>none</em>. See also “LoopOptimization” (HDL Coder).</td>
</tr>
<tr>
<td><strong>MapPersistentVarsToRAM</strong></td>
<td>Map persistent arrays to RAM. The default is <em>off</em>. See also “MapPersistentVarsToRAM” (HDL Coder).</td>
</tr>
<tr>
<td><strong>OutputPipeline</strong></td>
<td>Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “OutputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td><strong>ResetType</strong></td>
<td>Suppress reset logic generation. The default is <em>default</em>, which generates reset logic. See also “ResetType” (HDL Coder).</td>
</tr>
<tr>
<td><strong>SharingFactor</strong></td>
<td>Number of functionally equivalent resources to map to a single shared resource. The default is 0. See also “Resource Sharing” (HDL Coder).</td>
</tr>
<tr>
<td><strong>VariablesToPipeline</strong></td>
<td><strong>Warning</strong> VariablesToPipeline is not recommended. Use coder.hdl.pipeline instead.</td>
</tr>
<tr>
<td>-------------------------</td>
<td>---------------------------------------------------------------------</td>
</tr>
<tr>
<td></td>
<td>Insert a pipeline register at the output of the specified MATLAB variable or variables. Specify the list of variables as a character vector, with spaces separating the variables.</td>
</tr>
</tbody>
</table>

**Restrictions**

- The DUT subsystem must be single-rate.
- Inputs cannot have non-discrete (constant or Inf) sample time.
- The following predefined System objects are supported for code generation when you use them in the MATLAB System block:
  - hdl.RAM
  - comm.HDLCRCDetector
  - comm.HDLCRCGenerator
  - comm.HDLRSDecoder
  - comm.HDLRSEncoder
  - dsp.DCBlocker
  - dsp.HDLComplexToMagnitudeAngle
  - dsp.HDLFFT
  - dsp.HDLIFFT
  - dsp.HDLNCO
- If you use a user-defined System object, it must support HDL code generation. For information about user-defined System objects and requirements for HDL code generation, see “HDL Code Generation for System Objects” (HDL Coder).

See also “Generate Code for User-Defined System Objects” (HDL Coder) and “HDL Code Generation for System Objects” (HDL Coder).

**Fixed-Point Conversion**

Design and simulate fixed-point systems using Fixed-Point Designer™.

Actual data type support depends on block implementation.
See Also
MATLAB Function

Topics
“System Identification for an FIR System Using MATLAB System Blocks” on page 15-759
“MATLAB System Block with Variable-Size Input and Output Signals” on page 15-762
“Using Buses with MATLAB System Blocks” on page 15-766
“Pulse Width Modulation Using MATLAB System Block” on page 15-838
“MATLAB System Block”
“What Are System Objects?” (MATLAB)

Introduced in R2013b
Memory

Output input from previous time step

Library: Simulink / Discrete
        HDL Coder / Discrete

Description

The Memory block holds and delays its input by one major integration time step. When placed in an iterator subsystem, it holds and delays its input by one iteration. This block accepts continuous and discrete signals. The block accepts one input and generates one output. Each signal can be a scalar, vector, matrix, or N-D array. If the input is non-scalar, the block holds and delays all elements of the input by the same time step.

You specify the block output for the first time step using the Initial condition parameter. Careful selection of this parameter can minimize unwanted output behavior. However, you cannot specify the sample time. This block’s sample time depends on the type of solver used, or you can specify to inherit it. The Inherit sample time parameter determines whether sample time is inherited or based on the solver.

Tip  Avoid using the Memory block when both these conditions are true:

- Your model uses the variable-step solver ode15s or ode113.
- The input to the block changes during simulation.

When the Memory block inherits a discrete sample time, the block is analogous to the Unit Delay block. However, the Memory block does not support state logging. If logging the final state is necessary, use a Unit Delay block instead.

Comparison with Similar Blocks

The Memory, Unit Delay, and Zero-Order Hold blocks provide similar functionality but have different capabilities. Also, the purpose of each block is different.

This table shows recommended usage for each block.
### Purpose of the Block

**Unit Delay**
Implement a delay using a discrete sample time that you specify. The block accepts and outputs signals with a discrete sample time.

- [sldemo_enginewc](#) (Compression subsystem)

**Memory on page 1-1278**
Implement a delay by one major integration time step. Ideally, the block accepts continuous (or fixed in minor time step) signals and outputs a signal that is fixed in minor time step.

- [sldemo_bounce](#)
- [sldemo_clutch](#) (Friction Mode Logic/Lockup FSM subsystem)

**Zero-Order Hold**
Convert an input signal with a continuous sample time to an output signal with a discrete sample time.

- [sldemo_radar_eml](#)
- [aero_dap3dof](#)

Each block has the following capabilities.

<table>
<thead>
<tr>
<th>Capability</th>
<th>Memory</th>
<th>Unit Delay</th>
<th>Zero-Order Hold</th>
</tr>
</thead>
<tbody>
<tr>
<td>Specification of initial condition</td>
<td>Yes</td>
<td>Yes</td>
<td>No, because the block output at time $t = 0$ must match the input value.</td>
</tr>
<tr>
<td>Specification of sample time</td>
<td>No, because the block can only inherit sample time from the driving block or the solver used for the entire model.</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Support for frame-based signals</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Support for state logging</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

---

1-1279
Bus Support

The Memory block is a bus-capable block. The input can be a virtual or nonvirtual bus signal subject to the following restrictions:

- **Initial condition** must be zero, a nonzero scalar, or a finite numeric structure.
- If **Initial condition** is zero or a structure, and you specify a **State name**, the input cannot be a virtual bus.
- If **Initial condition** is a nonzero scalar, you cannot specify a **State name**.

For information about specifying an initial condition structure, see “Specify Initial Conditions for Bus Signals”.

All signals in a nonvirtual bus input to a Memory block must have the same sample time, even if the elements of the associated bus object specify inherited sample times. You can use a Rate Transition block to change the sample time of an individual signal, or of all signals in a bus. See “Specify Bus Signal Sample Times” and Bus-Capable Blocks for more information.

You can use an array of buses as an input signal to a Memory block. You can specify the **Initial condition** parameter with:

- The value 0. In this case, all the individual signals in the array of buses use the initial value 0.
- An array of structures that specifies an initial condition for each of the individual signals in the array of buses.
- A single scalar structure that specifies an initial condition for each of the elements that the bus type defines. Use this technique to specify the same initial conditions for each of the buses in the array.

For details about defining and using an array of buses, see “Combine Buses into an Array of Buses”.

Ports

Input

Port_1 — Input signal
scalar | vector | matrix | N-D array

Input signal, specified as a scalar, vector, matrix, or N-D array. The input can be continuous or discrete, containing real, or complex values of any data type Simulink supports.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus

Output

Port_1 — Input delayed by one major integration time step
scalar | vector | matrix | N-D array

Output is the input from the previous time step.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus

Parameters

Main

Initial condition — Initial condition
0 (default) | scalar | vector | matrix | N-D array

Specify the output at the initial integration step. This value must be 0 when you do not use a built-in input data type.

Programmatic Use

Block Parameter: InitialCondition
Type: character vector
Values: scalar | vector
Default: 'off'

Inherit sample time — Inherit sample time
off (default) | on

Select to inherit the sample time from the driving block:

- If the driving block has a discrete sample time, the block inherits the sample time.
- If the driving block has a continuous sample time, selecting this check box has no effect. The sample time depends on the type of solver used for simulating the model.

When this check box is cleared, the block sample time depends on the type of solver used for simulating the model:

- If the solver is a variable-step solver, the block sample time is continuous but fixed in minor time step: [0, 1].
- If the solver is a fixed-step solver, the [0, 1] sample time converts to the solver step size after sample-time propagation.

Programmatic Use
Block Parameter: InheritSampleTime
Type: character vector
Values: 'off' | 'on'
Default: 'off'

Direct feedthrough of input during linearization — Output the input during linearization and trim
off (default) | on

Select to output the input during linearization and trim. This selection sets the block mode to direct feedthrough.

Selecting this check box can cause a change in the ordering of states in the model when using the functions linmod, dlinmod, or trim. To extract this new state ordering, use the following commands.

First compile the model using the following command, where model is the name of the Simulink model.

```matlab
[sizes, x0, x_str] = model([],[],[],'lincompile');
```

Next, terminate the compilation with this command.
model([],[],[],'term');

The output argument, \texttt{x\_str}, which is a cell array of the states in the Simulink model, contains the new state ordering. When passing a vector of states as input to the \texttt{linmod}, \texttt{dlinmod}, or \texttt{trim} functions, the state vector must use this new state ordering.

**Programmatic Use**

**Block Parameter:** \texttt{LinearizeMemory}

**Type:** character vector

**Values:** 'off' | 'on'

**Default:** 'off'

*Treat as a unit delay when linearizing with discrete sample time* — *Linearize to unit delay for discrete inputs*

off (default) | on

Select to linearize the Memory block to a unit delay when the Memory block is driven by a signal with a discrete sample time.

**Programmatic Use**

**Block Parameter:** \texttt{LinearizeAsDelay}

**Type:** character vector

**Values:** 'off' | 'on'

**Default:** 'off'

**State Attributes**

**State name — Unique name for block state**

' ' (default) | alphanumeric string

Use this parameter to assign a unique name to the block state. The default is ' '. When this field is blank, no name is assigned. When using this parameter, remember these considerations:

- A valid identifier starts with an alphabetic or underscore character, followed by alphanumeric or underscore characters.
- The state name applies only to the selected block.

This parameter enables **State name must resolve to Simulink signal object** when you click **Apply**.
For more information, see “Apply Storage Classes to Individual Signal, State, and Parameter Data Elements” (Simulink Coder).

**Programmatic Use**

**Block Parameter:** StateName  
**Type:** character vector  
**Values:** unique name  
**Default:** ''

State name must resolve to Simulink signal object — Require state name resolve to a signal object  
off (default) | on

Select this check box to require that the state name resolves to a Simulink signal object.

**Dependencies**

To enable this parameter, specify a value for State name. This parameter appears only if you set the model configuration parameter Signal resolution to a value other than None.

Selecting this check box disables Code generation storage class.

**Programmatic Use**

**Block Parameter:** StateMustResolveToSignalObject  
**Type:** character vector  
**Values:** 'off' | 'on'  
**Default:** 'off'

Signal object class — Custom storage class package name  
Simulink.Signal (default) | <StorageClass.PackageName>

Choose a custom storage class package by selecting a signal object class that the target package defines. For example, to apply custom storage classes from the built-in package mpt, select mpt.Signal. Unless you use an ERT-based code generation target with Embedded Coder, custom storage classes do not affect the generated code.

If the class that you want does not appear in the list, select Customize class lists. For instructions, see “Target Class Does Not Appear in List of Signal Object Classes” (Embedded Coder).

For information about storage classes, see “Apply Storage Classes to Individual Signal, State, and Parameter Data Elements” (Simulink Coder). For information about custom storage classes, see “Apply Built-In and Customized Storage Classes to Data Elements” (Embedded Coder).
**Programmatic Use**

**Block Parameter:** StateSignalObject  
**Type:** character vector  
**Values:** 'Simulink.Signal' | '<StorageClass.PackageName>'  
**Default:** 'Simulink.Signal'

**Code generation storage class — State storage class for code generation**

Auto (default) | Model default | ExportedGlobal | ImportedExtern | ImportedExternPointer | BitField (Custom) | Model default | ExportToFile (Custom) | ImportFromFile (Custom) | FileScope (Custom) | AutoScope (Custom) | Struct (Custom) | GetSet (Custom) | Reusable (Custom)

Select state storage class for code generation.

- **Auto** is the appropriate storage class for states that you do not need to interface to external code.
- **StorageClass** applies the storage class or custom storage class that you select from the list. For information about storage classes, see “Apply Storage Classes to Individual Signal, State, and Parameter Data Elements” (Simulink Coder). For information about custom storage classes, see “Apply Built-In and Customized Storage Classes to Data Elements” (Embedded Coder).

Use **Signal object class** to select custom storage classes from a package other than Simulink.

**Dependencies**

To enable this parameter, specify a value for **State name**.

**Programmatic Use**

**Block Parameter:** StateStorageClass  
**Type:** character vector  
**Values:** 'Auto' | 'Model default' | 'ExportedGlobal' | 'ImportedExtern' | 'ImportedExternPointer' | 'Custom' | ...  
**Default:** 'Auto'

**TypeQualifier — Storage type qualifier**

'' (default) | const | volatile | ...

Specify a storage type qualifier such as const or volatile.
**Note** *TypeQualifier* will be removed in a future release. To apply storage type qualifiers to data, use custom storage classes and memory sections. Unless you use an ERT-based code generation target with Embedded Coder, custom storage classes and memory sections do not affect the generated code.

During simulation, the block uses the following values:

- The initial value of the signal object to which the state name is resolved
- Minimum and Maximum values of the signal object

For more information, see “Data Objects”.

**Dependencies**

To enable this parameter, set *Code generation storage class* to *ExportedGlobal*, *ImportedExtern*, *ImportedExternPointer*, or *Model default*. This parameter is hidden unless you previously set its value.

**Programmatic Use**

**Block Parameter:** `RTWStateStorageTypeQualifier`

**Type:** character vector

**Values:** `' ' | 'const' | 'volatile' | ...

**Default:** `' '"

**Block Characteristics**

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>bus</th>
<th>double</th>
<th>enumerated</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Direct Feedthrough</strong></td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Multidimensional Signals</strong></td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Variable-Size Signals</strong></td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Zero-Crossing Detection</strong></td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.
HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture
This block has a single, default HDL architecture.

HDL Block Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConstrainedOutputPipeline</td>
<td>Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. For more details, see “ConstrainedOutputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td>InputPipeline</td>
<td>Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “InputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td>OutputPipeline</td>
<td>Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “OutputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td>ResetType</td>
<td>Suppress reset logic generation. The default is default, which generates reset logic. See also “ResetType” (HDL Coder).</td>
</tr>
</tbody>
</table>

Complex Data Support
This block supports code generation for complex signals.
**PLC Code Generation**
Generate Structured Text code using Simulink® PLC Coder™.

**Fixed-Point Conversion**
Design and simulate fixed-point systems using Fixed-Point Designer™.

**See Also**
Unit Delay | Zero-Order Hold

**Topics**
“Apply Storage Classes to Individual Signal, State, and Parameter Data Elements” (Simulink Coder)
“Apply Built-In and Customized Storage Classes to Data Elements” (Embedded Coder)
“Data Objects”

**Introduced before R2006a**
Merge

Combine multiple signals into single signal

Library: Simulink / Signal Routing

Description

The Merge block combines inputs into a single output. The output value at any time is equal to the most recently computed output of its driving blocks. Specify the number of inputs by setting the parameter Number of inputs parameter.

Use Merge blocks to interleave input signals that update at different times into a combined signal in which the interleaved values retain their separate identities and times. To combine signals that update at the same time into an array or matrix signal, use a Concatenate block.

Guidelines for Using the Merge Block

When you use the Merge block, follow these guidelines:

- Always use conditionally executed subsystems to drive Merge blocks.
- Ensure that at most one of the driving conditionally executed subsystems executes at any time step.
- Ensure that all input signals have the same sample time.
- Do not branch a signal that inputs to a Merge block, if you use the default setting of Classic for the Model Configuration Parameters > Diagnostics > Underspecified initialization detection parameter.
- For all conditionally executed subsystem Outport blocks that drive Merge blocks, set the Output when disabled parameter to held.
- If the output of a Model block is coming from a MATLAB Function block or a Stateflow chart, do not connect that output port to the input port of the Merge block.

For each input of a Merge block, the topmost nonatomic and nonvirtual source must be a conditionally executed subsystem that is not an Iterator Subsystem.
The next diagram shows valid Merge block usage, merging signals from two conditionally executed subsystems.

![Diagram of valid Merge block usage](image)

**Bus Support**

The Merge block is a bus-capable block. The inputs can be virtual or nonvirtual bus signals subject to these restrictions:

- The number of inputs must be greater than one.
- **Initial output** must be zero, a nonzero scalar, or a finite numeric structure.
- **Allow unequal port widths** must be disabled.
- All inputs to the merge must be buses and must be equivalent (same hierarchy with identical names and attributes for all elements).

All signals in a nonvirtual bus input to a Merge block must have the same sample time. You can use a Rate Transition block to change the sample time of an individual signal, or of all signals in a bus.

**Merging S-Function Outputs**

The Merge block can merge a signal from an S-Function block only if the memory used to store the output from the S-Function is reusable. Simulink software displays an error message if you attempt to update or simulate a model that connects a nonreusable port of an S-Function block to a Merge block. See `ssSetOutputPortOptimOpts`. 

1-1290
Limitations

- All signals that connect to a Merge block, are functionally the same signal. Therefore, they are subject to the restriction that a given signal can have at most one associated signal object. See Simulink.Signal for more information.

- Run-time diagnostics do not run if the inputs to a merge block are from a single initiator. For example, a single initiator could be a Stateflow chart executing function-call subsystems that are connected to a Merge block.

- Do not set the outports of conditionally executed subsystems being merged to reset when disabled. This action can cause multiple subsystems to update the block at the same time. Specifically, the disabled subsystem updates the Merge block by resetting its output, while the enabled subsystem updates the block by computing its output.

To prevent this behavior, set the Outport block parameter Output when disabled to held for each conditionally executed subsystem being merged.

**Note** If you are using Simplified Initialization Mode, set the Outport block parameter Output when disabled to held.

- A Merge block does not accept input signals whose elements have been reordered or partially selected. In addition, do not connect input signals to the block that have been combined outside of a conditionally executed subsystem.

You can use an array of buses as an input signal to a Merge block with these limitations:

- **Allow unequal port widths** — Clear this parameter.
- **Initial condition** — You can specify this parameter using:
  
  - The value 0. In this case, each of the individual signals in the array of buses use the initial value 0.
  
  - An array of structures that specifies an initial condition for each of the individual signals in the array of buses.
  
  - A single scalar structure that specifies an initial condition for each of the elements that the bus type defines. Use this technique to specify the same initial conditions for each of the buses in the array.
Ports

Input

Port_1 — First input signal
scalar | vector

First input signal merged with the other input signals.
Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus

Port_n — nth input signal
scalar | vector

nth input signal merged with the other input signals.
Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus

Output

Port_1 — Output signal
scalar | vector

Output signal merged from the input signals.
Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus

Parameters

Number of inputs — Number of input signals to merge
2 (default) | integer

Specify the number of input signals to merge. The block creates a port for each input signal.

Programmatic Use
Block Parameter: Inputs
**Type:** character vector  
**Values:** integer  
**Default:** '2'

**Initial output — Initial output value**

[ ] (default) | scalar | vector

Specify the initial value of the output signal. If you do not specify an initial output value, then initial output depends on the initialization mode and the driving blocks.

In Simplified initialization mode, for an unspecified (empty matrix []) value of **Initial output**, the block uses the default initial value of the output data type. For information on the default initial value, see “Initializing Signal Values”. In Classic initialization mode, for an unspecified (empty matrix []) value of **Initial output**, the initial output of the block equals the most recently evaluated initial output of the driving blocks. Since the initialization ordering for these sources can vary, initialization can be inconsistent for the simulation and the code generation of a model.

**Programmatic Use**  
**Block Parameter:** InitialOutput  
**Type:** character vector  
**Values:** scalar | vector  
**Default:** '[ ]'

**Allow unequal port widths — Allow inputs of unequal dimensions**

off (default) | on

Select this check box to allow the block to accept inputs having different numbers of elements. The block allows you to specify an offset for each input signal relative to the beginning of the output signal. The width of the output signal is

\[
\max (w_1+o_1, w_2+o_2, \ldots, w_n+o_n)
\]

where \( w_1, \ldots, w_n \) are the widths of the input signals and \( o_1, \ldots, o_n \) are the offsets for the input signals.

If you clear this check box, the Merge block accepts only inputs of equal dimensions and outputs a signal of the same dimensions as the inputs.

**Programmatic Use**  
**Block Parameter:** AllowUnequalInputPortWidths  
**Type:** character vector

1-1293
Values: 'off' | 'on'
Default: 'off'

**Input port offsets — Offset for input signals**

[ ] (default) | vector

Enter a vector to specify the offset of each input signal relative to the beginning of the output signal.

**Programmatic Use**

**Block Parameter:** InputPortOffsets

**Type:** character vector

**Values:** scalar | vector

**Default:** '[ ]'

---

**Block Characteristics**

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>bus</th>
<th>double</th>
<th>enumerated</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
<th>string</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Direct Feedthrough</strong></td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Multidimensional Signals</strong></td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Variable-Size Signals</strong></td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Zero-Crossing Detection</strong></td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

**Extended Capabilities**

**C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.
In the code generation workflow, when the Merge block receives a constant value and non-constant sample times, one of these conditions must hold. Otherwise Simulink displays an error.

- The source of the constant value is a grounded signal.
- The source of the constant value is a constant block with a non-tunable parameter.
  - There is only one constant block that feeds the Merge block.
  - All other input signals to the Merge block are from conditionally executed subsystems.
  - The Merge block and outport blocks of all conditionally executed subsystems does not specify any initial outputs.

**Fixed-Point Conversion**
Design and simulate fixed-point systems using Fixed-Point Designer™.

**See Also**
Manual Switch | Switch

**Topics**
“Conditionally Executed Subsystems Overview”

**Introduced before R2006a**
**Entity Transport Delay**

Introduce a delay in propagation of a SimEvents message

**Library:** Simulink / Continuous

SimEvents

**Description**

The Entity Transport Delay block delays an entity for a period of time, named *transport delay*. The first input is the entity that is transported from point A to point B on a constant-length moving surface whose speed changes over time. The value from the second input is the *instantaneous delay*. The speed of the surface is the reciprocal of instantaneous delay. The product of instantaneous delay and the speed is 1.

The block calculates the implemented transport delay by the distance-speed-time relationship, where the surface length (distance) is equal to the integral of the variable surface speed over the duration of the transport delay (time). For more information about this calculation, see Variable Transport Delay.

The Entity Transport Delay block connects SimEvents to Simulink using the input from a Simulink signal and computing the transport delay as a continuous process, and applying this delay to an entity in a discrete-event process. For an example that uses the Entity Transport Delay block, see “Modeling Cyber-Physical Systems” on page 15-840.

**Ports**

**Input**

**Port 1 — Input entity**

SimEvents entity

SimEvents entity or message. For more information on entities in SimEvents, see “Entities in a SimEvents Model” (SimEvents).
**t_i — Instantaneous delay**

scalar | vector | matrix

Instantaneous delay in the transport of the SimEvents entity. \( t_i \) is the reciprocal of the speed of the entity. For more information on the calculation of instantaneous delay, see “Variable Transport Delay” on page 1-2263.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | bus

**Output**

**Port_1 — Delayed entity**

SimEvents entity

SimEvents entity with the instantaneous delay \( t_i \) applied to it.

**n — Number of delayed entities**

real scalar

Secondary output signal of the block, which displays the number of entities processed in a time step.

**Dependencies**

To enable this port, enable the **Output number of entities, n** parameter

Data Types: double

**Parameters**

**Capacity — Specify the capacity of the block**

inf (default) | scalar

Specify capacity to accept entities to be delayed.

**Programmatic Use**

**Block Parameter:** Capacity  
**Type:** character vector  
**Values:** 'inf' | real scalar  
**Default:** 'inf'
Output number of entities in block, n — Outputs the number of delayed entities present in the block
off (default) | on

Number of entities present in the block that are being delayed.

Programmatic Use
Block Parameter: ShowNumberEntitiesInBlock
Type: character vector
Values: 'on' | 'off'
Default: 'off'

Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>bus</th>
<th>double</th>
<th>enumerated</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also
Receive | Send | Transport Delay | Variable Transport Delay
Topics
“Message-Based Communication Between Software Components”

Introduced in R2019b
Receive, Message Receive

Extract data from received messages
Library: Simulink / Messages & Events
        SimEvents

Description

The Send block extracts data from received messages and writes them to the output signal port. If there are no new messages when the block executes, the block uses the Value source when queue is empty value. Receive and Message Receive blocks are identical blocks.

• Select Hold last value to hold data read from the last message.
• Select Use initial value to write default data.

Ports

Input

Port_1 — Input message
scalar | vector | matrix

The input port for the message.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 |
Boolean | enumerated | bus | fixed point

Output

Port_S — Show whether a message was received
scalar

Outputs 1 if the block receives a message successfully, and 0 otherwise.
Dependencies

To enable this port, select the check box labeled Show receive status.

Data Types: double

Port 1 — Output signal

scalar | vector | matrix

Output port for the signal.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | enumerated | bus | fixed point

Parameters

Use internal queue — Select to use an internal queue

on (default for SimEvents) | off (default for Simulink)

Select this check box if you use an internal queue to receive messages.

Programmatic Use

Block Parameter: UseInternalQueue
Type: character vector
Values: 'on' | 'off'
Default: 'on'

Overwrite the oldest element if queue is full — Specify queue overwriting policy

on (default for Simulink) | off (default for SimEvents)

Select this check box to choose between two queue overwriting policies.

• If you select the check box, an incoming message overwrites the oldest if the queue is full.
• If you clear the check box, the block does not accept new messages if the queue is full.

Dependencies

This parameter is visible when you select the box labeled Use internal queue.
Programmatic Use
Block Parameter: QueueOverwriting
Type: character vector
Values: 'on' | 'off'
Default: 'on' (for Simulink) and 'off' (for SimEvents)

Queue length — Specify the length of the message queue
16 (default) | scalar
Specify message queue capacity.

Dependencies
This parameter is visible when you select the box labeled Use internal queue.

Programmatic Use
Block Parameter: QueueLength
Type: character vector
Values: '16' | scalar
Default: '16'

Queue type — Specify message queue sorting policy
FIFO (default) | LIFO | Priority
The block supports three message sorting policies:

- First-in-first-out (FIFO) — The oldest message in the storage departs first.
- Last-in-first-out (LIFO) — The newest message in the storage departs first.
- Priority — Messages or entities are sorted based on their priority

The priority queue can be used only when the Overwrite the oldest element if queue is full check box is cleared.

Note  Priority queue does not support fixed point data type.

Dependencies
This parameter is visible when you select the box labeled Use internal queue.

Programmatic Use
Block Parameter: QueueType
Type: character vector
Values: 'FIFO' | 'LIFO' | 'Priority'
Default: 'FIFO'

Priority order — Specify message queue priority
Ascending (default) | Descending

Choose the direction of sorting messages based on priority.

- Ascending — Messages with smaller priority values appear in front of the queue.
- Descending — Messages with greater priority values appear in front of the queue.

Dependencies

This parameter is visible when you select the box labeled Use internal queue and Queue type > Priority.

Programmatic Use
Block Parameter: PriorityOrder
Type: character vector
Values: 'Ascending' | 'Descending'
Default: 'Ascending'

Show receive status — Show whether a message was received
off (default) | on

Select this check box to show whether a message was received. If this check box is selected the block outputs 1 if it receives a message successfully, and 0 otherwise.

Programmatic Use
Block Parameter: ShowQueueStatus
Type: character vector
Values: 'on' | 'off'
Default: 'off'

Initial value — Set initial data value
[] (unspecified) (default) | scalar

Enter an initial data value for the queue before the arrival of the first message. The default value [] (unspecified) is treated as 0 with data type double.

Programmatic Use
Block Parameter: InitialValue
Type: character vector
Values: '[]' | scalar
Default: '[]'

Value source when queue is empty — Value source for empty queue
Hold last value (default) | Use initial value

Specify the value to receive when received message queue is empty.
- Hold last value (default) — Holds data read from the last message.
  Initially, if the block tries to receive a message and fails, it outputs the initial value.
- Use initial value — Writes default data.

Dependencies
This parameter is visible when you select the box labeled Use internal queue.

Programmatic Use
Block Parameter: ValueSourceWhenQueueIsEmpty
Type: character vector
Values: 'Hold last value' | 'Use initial value'
Default: 'Hold last value'

Sample time (-1 for inherited) — Specify the time interval between samples
-1 (default) | scalar

To inherit the sample time, set this parameter to -1. See “Specify Sample Time” for more information.

Programmatic Use
Block Parameter: SampleTime
Type: character vector
Values: '-1' | scalar
Default: '-1'

Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>bus</th>
<th>double</th>
<th>enumerated</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
<th>string</th>
</tr>
</thead>
<tbody>
<tr>
<td>Feature</td>
<td>Value</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-------------------------</td>
<td>-------</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Extended Capabilities**

**C/C++ Code Generation**
Generate C and C++ code using Simulink® Coder™.

**See Also**
Send

*Introduced in R2016a*
Send, Message Send

Create and send message

**Library:** Simulink / Messages & Events
SimEvents

Description

The Send block reads the value of an input signal, and sends a message that carries this value. In message-based communication, a message is a discrete-item of interest that carry data of any type that Simulink supports. Send and Message Send blocks are identical blocks.

Ports

Input

**Port_Enable — External enable signal**
scalar

Input port to enable the block to send a message. For any input value that is greater than 0 send is enabled. For any value less than or equal to 0, the send is disabled.

Dependencies

To enable this port, select the box labeled **Show enable port**.

Data Types: double

**Port_1 — Input signal**
scalar | vector | matrix

This block accepts inputs of any type that Simulink supports, including enumerated types and converts the input signal to a message. For more information, see “Data Types Supported by Simulink”.

1-1306
Output

**Port_1 — Output message**

scalar | vector | matrix

The block outputs a message with constant priority 20.

Parameters

**Show enable port — Display the enable port**

off (default) | on

Select this check box to display enable port.

**Programmatic Use**

**Block Parameter:** ShowEnablePort

**Type:** character vector

**Values:** 'on' | 'off'

**Default:** 'off'

Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>bus</th>
<th>double</th>
<th>enumerated</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
<th>string</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also
Receive

Introduced in R2016a
MinMax

Output minimum or maximum input value

Library:
- Simulink / Math Operations
- HDL Coder / Math Operations

Description

The MinMax block outputs either the minimum or the maximum element or elements of the inputs. You choose whether the block outputs the minimum or maximum values by setting the Function parameter.

The MinMax block ignores any input value that is NaN, except when every input value is NaN. When all input values are NaN, the output is NaN, either as a scalar or the value of each output vector element.

Ports

Input

Port 1 — Input signal
scalar | vector | matrix

Provide an input signal from which the block outputs the maximum or minimum values.

- When the block has one input port, the input must be a scalar or a vector. The block outputs a scalar equal to the minimum or maximum element of the input vector.
- When the block has multiple input ports, all nonscalar inputs must have the same dimensions. The block expands any scalar inputs to have the same dimensions as the nonscalar inputs. The block outputs a signal having the same dimensions as the input. Each output element equals the minimum or maximum of the corresponding input elements.
Dependencies

To support matrix input, you must set the **Number of input ports** parameter to an integer greater than one. All nonscalar inputs must have the same dimensions.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point

Port_N — N-th input signal
scalar | vector | matrix

Provide an input signal from which the block outputs the maximum or minimum values.

When the block has multiple input ports, all nonscalar inputs must have the same dimensions. The block expands any scalar inputs to have the same dimensions as the nonscalar inputs. The block outputs a signal having the same dimensions as the input. Each output element equals the minimum or maximum of the corresponding input elements.

Dependencies

To provide more than one input signal, set the **Number of input ports** to an integer greater than 1.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point

Output

Port_1 — Minimum or maximum values of inputs
scalar | vector | matrix

When the block has one input, the output is a scalar value, equal to the minimum or maximum of the input elements. When the block has multiple inputs, the output is a signal having the same dimensions as the input. Each output element equals the minimum or maximum of the corresponding input elements.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point
Parameters

Main

Function — Specify minimum or maximum
min (default) | max

Specify whether to apply the function min or max to the input.

Programmatic Use
Block Parameter: Function
Type: character vector
Values: 'min' | 'max'
Default: 'min'

Number of input ports — Specify number of input ports
1 (default) | positive integer

Specify the number of inputs to the block.

Programmatic Use
Block Parameter: Inputs
Type: character vector
Values: positive integer
Default: '1'

Enable zero-crossing detection — Enable zero-crossing detection
on (default) | Boolean

Select to enable zero-crossing detection. For more information, see “Zero-Crossing Detection” in the Simulink documentation.

Programmatic Use
Block Parameter: ZeroCross
Type: character vector, string
Values: 'off' | 'on'
Default: 'on'

Sample time — Specify sample time as a value other than -1
-1 (default) | scalar | vector
Specify the sample time as a value other than -1. For more information, see “Specify Sample Time”.

**Dependencies**

This parameter is not visible unless it is explicitly set to a value other than -1. To learn more, see “Blocks for Which Sample Time Is Not Recommended”.

**Programmatic Use**

**Block Parameter:** SampleTime  
**Type:** character vector  
**Values:** scalar or vector  
**Default:** '-1'

**Signal Attributes**

**Require all inputs to have the same data type — Inputs must have the same data type**

*off* (default) | *on*

Select this check box to require that all inputs have the same data type.

**Programmatic Use**

**Block Parameter:** Inputs  
**Type:** character vector  
**Values:** 'off' | 'on'  
**Default:** 'off'

**Output minimum — Minimum output value for range checking**

[ ] (default) | scalar

Lower value of the output range that Simulink checks.

Simulink uses the minimum to perform:

- Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters”) for some blocks.
- Simulation range checking (see “Signal Ranges” and “Enable Simulation Range Checking”).
- Automatic scaling of fixed-point data types.
- Optimization of the code that you generate from the model. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL
or external mode. For more information, see “Optimize using the specified minimum and maximum values” (Simulink Coder).

**Note** Output minimum does not saturate or clip the actual output signal. Use the Saturation block instead.

**Programmatic Use**  
**Block Parameter:** OutMin  
**Type:** character vector  
**Values:** '[' ']'
**Default:** '[' ']

**Output maximum — Maximum output value for range checking**  
[] (default) | scalar

Upper value of the output range that Simulink checks.

Simulink uses the maximum value to perform:

- Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters”) for some blocks.
- Simulation range checking (see “Signal Ranges” and “Enable Simulation Range Checking”).
- Automatic scaling of fixed-point data types.
- Optimization of the code that you generate from the model. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. For more information, see “Optimize using the specified minimum and maximum values” (Simulink Coder).

**Note** Output maximum does not saturate or clip the actual output signal. Use the Saturation block instead.

**Programmatic Use**  
**Block Parameter:** OutMax  
**Type:** character vector  
**Values:** '[' ']'
**Default:** '[' ']

1-1313
Output data type — Specify the output data type
Inherit: Inherit via internal rule (default) | Inherit: Inherit via back propagation | double | single | int8 | uint8 | int16 | uint16 | int32 | uint32 | int64 | uint64 | fixdt(1,16,0) | fixdt(1,16,2^0,0) | <data type expression>

Choose the data type for the output. The type can be inherited, specified directly, or expressed as a data type object such as Simulink.NumericType.

Click the **Show data type assistant** button to display the Data Type Assistant, which helps you set the data type attributes. For more information, see “Specify Data Types Using Data Type Assistant”.

Programmatic Use

**Block Parameter:** OutDataTypeStr

**Type:** character vector

**Values:** 'Inherit: Inherit via internal rule' | 'Inherit: Inherit via back propagation' | 'single' | 'int8' | 'uint8' | 'int16' | 'uint16' | 'int32' | 'uint32' | 'int64' | 'uint64' | 'fixdt(1,16,0)' | 'fixdt(1,16,2^0,0)' | '<data type expression>'

**Default:** 'Inherit: Inherit via internal rule'

Lock output data type setting against changes by the fixed-point tools — Prevent fixed-point tools from overriding Output data type

off (default) | on

Select this parameter to prevent the fixed-point tools from overriding the Output data type you specify on the block. For more information, see “Use Lock Output Data Type Setting” (Fixed-Point Designer).

Programmatic Use

**Block Parameter:** LockScale

**Type:** character vector

**Values:** 'off' | 'on'

**Default:** 'off'

Integer rounding mode — Rounding mode for fixed-point operations

Floor (default) | Ceiling | Convergent | Nearest | Round | Simplest | Zero

Specify the rounding mode for fixed-point operations. For more information, see “Rounding” (Fixed-Point Designer).
Block parameters always round to the nearest representable value. To control the rounding of a block parameter, enter an expression using a MATLAB rounding function into the mask field.

**Programmatic Use**

**Block Parameter:** RndMeth

**Type:** character vector

**Values:** 'Ceiling' | 'Convergent' | 'Floor' | 'Nearest' | 'Round' | 'Simplest' | 'Zero'

**Default:** 'Floor'

**Saturate on integer overflow — Method of overflow action**

off (default) | on

Specify whether overflows saturate or wrap.

<table>
<thead>
<tr>
<th>Action</th>
<th>Rationale</th>
<th>Impact on Overflows</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Select this check box (on).</td>
<td>Your model has possible overflow, and you want explicit saturation protection in the generated code.</td>
<td>Overflows saturate to either the minimum or maximum value that the data type can represent.</td>
<td>The maximum value that the int8 (signed, 8-bit integer) data type can represent is 127. Any block operation result greater than this maximum value causes overflow of the 8-bit integer. With the check box selected, the block output saturates at 127. Similarly, the block output saturates at a minimum output value of -128.</td>
</tr>
</tbody>
</table>
**Action** | **Rationale** | **Impact on Overflows** | **Example**  
--- | --- | --- | ---  
Do not select this check box (off). | You want to optimize efficiency of your generated code. You want to avoid overspecifying how a block handles out-of-range signals. For more information, see “Check for Signal Range Errors”. | Overflows wrap to the appropriate value that is representable by the data type. | The maximum value that the int8 (signed, 8-bit integer) data type can represent is 127. Any block operation result greater than this maximum value causes overflow of the 8-bit integer. With the check box cleared, the software interprets the overflow-causing value as int8, which can produce an unintended result. For example, a block result of 130 (binary 1000 0010) expressed as int8, is -126.  

When you select this check box, saturation applies to every internal operation on the block, not just the output, or result. Usually, the code generation process can detect when overflow is not possible. In this case, the code generator does not produce saturation code.**

**Programmatic Use**

**Block Parameter:** SaturateOnIntegerOverflow  
**Type:** character vector  
**Values:** 'off' | 'on'  
**Default:** 'off'

## Block Characteristics

<table>
<thead>
<tr>
<th><strong>Data Types</strong></th>
<th>**Boolean</th>
<th>double</th>
<th>fixed point</th>
<th>integer</th>
<th>single**</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Direct Feedthrough</strong></td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Multidimensional Signals</strong></td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Variable-Size Signals | yes
---|---
Zero-Crossing Detection | yes

### Extended Capabilities

**C/C++ Code Generation**
Generate C and C++ code using Simulink® Coder™.

**HDL Code Generation**
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

**HDL Architecture**

This block has multi-cycle implementations that introduce additional latency in the generated code. To see the added latency, view the generated model or validation model. See “Generated Model and Validation Model” (HDL Coder).

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Additional cycles of latency</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>default Tree</td>
<td>0</td>
<td>Generates a tree structure of comparators.</td>
</tr>
<tr>
<td>Cascade</td>
<td>1, when block has a single vector input port.</td>
<td>This implementation is optimized for latency * area, with medium speed. See “Cascade Architecture Best Practices” (HDL Coder).</td>
</tr>
</tbody>
</table>
### HDL Block Properties

<table>
<thead>
<tr>
<th>General</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ConstrainedOutputPipeline</strong></td>
<td>Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. For more details, see “ConstrainedOutputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td><strong>InputPipeline</strong></td>
<td>Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “InputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td><strong>OutputPipeline</strong></td>
<td>Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “OutputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td><strong>InstantiateStages</strong></td>
<td>Generate a VHDL entity or Verilog module for each cascade stage. The default is off. See also “InstantiateStages” (HDL Coder).</td>
</tr>
<tr>
<td><strong>SerialPartition</strong></td>
<td>Specify partitions for Cascade-serial implementations as a vector of the lengths of each partition. The default setting uses the minimum number of stages. See also “SerialPartition” (HDL Coder).</td>
</tr>
</tbody>
</table>

**Note** To enable the **LatencyStrategy** setting for the MinMax block, you must specify **Tree** as the **HDL Architecture**.

### Native Floating Point

| **LatencyStrategy** | Specify whether to map the blocks in your design to inherit, Max, Min, or Zero for the floating-point operator. The default is inherit. See also “LatencyStrategy” (HDL Coder). |

### PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.
Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

See Also
MinMax Running Resettable

Introduced before R2006a
MinMax Running Resettable

Determine minimum or maximum of signal over time

**Library:** Simulink / Math Operations

---

**Description**

The MinMax Running Resettable block outputs the minimum or maximum of all past inputs \( u \). You specify whether the block outputs the running minimum or maximum with the **Function** parameter.

The block can reset its state based on an external reset signal \( R \). When the reset signal \( R \) is nonzero (true), the block resets the output to the value of the **Initial condition** parameter.

The input can be a scalar, vector, or matrix signal. The block outputs a signal having the same dimensions as the input. Each output element equals the running minimum or maximum of the corresponding input elements.

---

**Ports**

**Input**

**u — Input signal**

scalar | vector | matrix

Input signal as a scalar, vector, or matrix. Based on what you specify for the **Function** parameter, the block outputs the minimum or maximum value of all past inputs \( u \).

If you specify a scalar value for the **Initial condition** parameter, the block expands the parameter to have the same dimensions as nonscalar input \( u \).
**Data Types:** single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | bus

**R — Reset signal**
scalar | vector | matrix

The input port accepting the reset signal as a scalar, vector, or matrix. When the reset signal is nonscalar, it must have the same dimensions as input signal u. As long as the reset signal has a value of zero, the block outputs the running minimum or maximum value of input u. Whenever the reset signal has a nonzero value (true), the block resets the output to the value of the **Initial condition** parameter.

**Output**

**y — Running minimum or maximum value**
scalar | vector | matrix

Output signal, specified as a scalar, vector, or matrix, where each output element equals the running minimum or maximum value of the corresponding input elements. Output signal y has the same data type and dimensions as input signal u.

When the block receives a nonzero (true) reset signal, the block resets the output to the value of the **Initial condition** parameter.

**Parameters**

**Function — Specify minimum or maximum**
min (default) | max

Specify whether the block outputs the running minimum or maximum value of the corresponding input elements.

**Programmatic Use**

**Block Parameter:** Function
Type: character vector
Values: 'min' | 'max'
Default: 'min'

Initial condition — Value to reset output to
0.0 (default) | scalar or vector

Specify the initial condition value. When the reset input signal \( R \) is true, the block resets the output to the value you specify.

Programmatic Use
Block Parameter: \( \text{vinit} \)
Type: character vector
Values: scalar or vector
Default: '0.0'

Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>double</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.
PLC Code Generation
Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

See Also
MinMax

Introduced before R2006a
Model

Reference another model to create model hierarchy

Library: Simulink / Ports & Subsystems
HDLCoder / Ports & Subsystems

Description

The Model block allows you to include a model as a block in another model. The included model is called a *referenced model*, and the model containing it (using the Model block) is called the *parent model*.

The Model block displays input and output ports corresponding to the top-level input and output ports of the referenced model. Using these ports allow you to connect the referenced model to other blocks in the parent model. See “Model References” for more information.

By default, the contents of a referenced model are user-visible by double-clicking the Model block. However, you can hide the contents of a referenced model by making the model a protected model.

To set the referenced model and simulation parameters, open the **Block Parameters** dialog box and use the **Main** tab. To specify values for model arguments, use the **Arguments** tab.

Ports

Input

*Input_Port_1* — Input port corresponding to root-level Inport, Enable, and Trigger blocks of referenced model

real or complex values of any data type supported by Simulink

The Model block has an input port for each root-level Inport, Enable, or Trigger block in the referenced model. The name of the Model block port matches the name of the
corresponding referenced model input block. The Model block input signals must be valid for the corresponding referenced model input blocks. See “Model Reference Interface and Boundary”.

Input signals can have real or complex values of any data type supported by Simulink, including bus objects, arrays of buses, fixed-point, and enumerated data types. For details about data types, see “Data Types Supported by Simulink”.

**Output**

**Output_Port_1** — Output port corresponding to root-level Outport block of referenced model
real or complex values of any data type supported by Simulink

The Model block has an output port for each root-level Outport block in the referenced model. The name of the Model block port matches the name of the corresponding Outport block. The Model block output signals are the signals from the corresponding referenced model Outport blocks. See “Model Reference Interface and Boundary”.

Output signals can have real or complex values of any data type supported by Simulink, including bus objects, arrays of buses, fixed-point, and enumerated data types. For details about data types, see “Data Types Supported by Simulink”.

**Parameters**

**Content Preview** — Display representation of block contents on the block
'on' (default) | 'off'

Display a representation of block contents on the block. For more information, see “Preview Content of Model Components”.

**Programmatic Use**

**Parameter:** ContentPreviewEnabled

**Type:** character vector

**Value:** 'on' | 'off'

**Default:** 'on'
Main Tab

Model name — File name of referenced model
' ' (default) | character vector

Path to the referenced model. The file name must be a valid MATLAB identifier. The extension, for example, .slx, is optional. The file name must contain fewer than 60 characters, exclusive of the .slx or .mdl suffix.

To navigate to the model that you want to reference, click Browse.

To view the model that you specified, click Open Model.

Programmatic Use
Parameter: ModelFile
Type: character vector
Value: ' ' | '<file name>'
Default: '

Simulation mode — Simulation mode for model reference
Normal (default) | Accelerator | Software-in-the-loop (SIL) | Processor-in-the-loop (PIL)

Specify the simulation mode for the Model block. The simulation mode for the Model block can be different than the simulation mode of its referenced model and of other models in the model hierarchy.

• Accelerator — Create a MEX-file for the referenced model and then executes the referenced model by running the S-function.
• Normal — Execute the referenced model interpretively, as if the referenced model is an atomic subsystem implemented directly within the parent model.
• Software-in-the-loop (SIL) — This option requires the Embedded Coder software. Generate production code based on the Code Interface parameter setting. The code is compiled for, and executed on, the host platform.
• Processor-in-the-loop (PIL) — This option requires the Embedded Coder software. Generate production code based on the Code Interface parameter setting. This code is compiled for, and executed on, the target platform. A documented target connectivity API supports exchange of data between the host and target at each time step during the PIL simulation.

The corners of the Model block reflect the simulation mode of the Model block. For normal mode, the corners have empty triangles. For accelerator mode, the corner
triangles are filled in. For SIL and PIL modes, the corners are filled in and the word (SIL) or (PIL) appears on the block icon.

While you can set a referenced model to rapid accelerator mode, simulation ignores the referenced model simulation mode. For information about simulation mode precedence in a model hierarchy, see “Choose Simulation Modes for Model Hierarchies”.

**Programmatic Use**

**Parameter**: SimulationMode  
**Type**: character vector  
**Value**: 'Normal' | 'Accelerator' | 'Software-in-the-loop' | 'Processor-in-the-loop'  
**Default**: 'Normal'

**Code interface — Generate code from top model or referenced model**

Model reference (default) | Top model

Specify whether to generate the code from the top model or the referenced model for SIL and PIL simulation modes. To deploy the generated code as part of a larger application that uses the referenced model, specify Model reference. To deploy the generated code as a standalone application, specify Top model.

Model reference  
Code generated from referenced model as part of a model hierarchy. Code generation uses the slbuild('model', 'ModelReferenceCoderTarget') command.

Top model  
Code generated from top model with the standalone code interface. Code generation uses the slbuild('model') command.

**Dependencies**

To display and enable this parameter, select either Software-in-the-loop (SIL) or Processor-in-the-loop (SIL) from the Simulation mode drop-down list.

**Programmatic Use**

**Parameter**: CodeInterface  
**Type**: character vector  
**Value**: 'Model reference' | 'Top model'  
**Default**: 'Model reference'

**Show model initialize port — Control display of initialize event port**

off (default) | on
Control display of initialize event port on the Model block.

- **off**
  - Remove port.
- **on**
  - Display model initialize event port.

**Programmatic Use**

**Block parameter:** ShowModelInitializePort

**Type:** character vector

**Value:** 'off' | 'on'

**Default:** 'off'

Show model reset port — Control display of reset event ports

- **off** (default) | **on**

Control display of reset event ports on the Model block.

- **off**
  - Remove port.
- **on**
  - Display model reset event ports.

**Programmatic Use**

**Block parameter:** ShowModelResetPorts

**Type:** character vector

**Value:** 'off' | 'on'

**Default:** 'off'

Show model terminate port — Control display of terminate event port

- **off** (default) | **on**

Control display of terminate event port on Model block.

- **off**
  - Remove port.
on
Display model block port.

Programmatic Use
Block parameter: ShowModelTerminatePort
Type: character vector
Value: 'off' | 'on'
Default: 'off'

Schedule rates — Control display of periodic event ports
off (default) | on
Control display of periodic event ports on Model block.

off
Hide ports.

on
Display ports for rate-based models. A rate-based model is a model with the Sample time for a connected Inport block specified.

If you want to manually specify the port rates, set the parameter AutoFillPortDiscreteRates to 'off', and then add the port rates to the parameter PortDiscreteRates.

Programmatic Use
Block parameter: ScheduleRates
Type: character vector
Value: 'off' | 'on'
Default: 'off'

Schedule rates with — Control display of periodic event ports
Ports (default) | Schedule Editor
Control display of periodic event ports on Model block.

Ports
Display ports for rate based models.

Schedule Editor
Create partitions for rate based models.
**Programmatic Use**

**Block parameter:** ScheduleRatesWith  
**Type:** character vector  
**Value:** 'Ports' | 'ScheduleEditor'  
**Default:** 'Ports'

**Instance Parameters Tab**

**Instance parameters — Display instance parameters and specify values for referenced model**

Number | Workspace variable | Mathematical expression | Structure or structure field

Display instance parameters and specify instance-specific values. Instance parameters are block parameters that have been configured in the referenced model to use a different value for each instance of the model.

To specify instance-specific values, use the **Value** column in the table. To specify that a parameter value can be overridden by the parent model, select the **Argument** check box. For more information about configuring instance parameters in a referenced model and specifying instance-specific values at the Model block, see “Parameterize Instances of a Reusable Referenced Model”.

When changing instance-specific values, you can use a partial structure, which has fields that correspond to only the instance-parameters whose values you want to change. Instance parameters not included in the partial structure retain their values. In the structure, include the instance parameter names and values, represented as character vectors.

**Programmatic Use**

**Block parameter:** ParameterArgumentValues  
**Type:** structure  
**Value:** structure  
**Default:** structure with no fields

**Block parameter:** InstanceParameters  
**Type:** structure array  
**Value:** structure with fields: `Name`, `Value`, `Path`, and `Argument`. 
Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>bus</th>
<th>double</th>
<th>enumerated</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
<th>string</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>limited</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>limited</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

a. Actual data type or capability support depends on block implementation.

Compatibility Considerations

Specifying variant models in Model blocks is no longer supported

Warns starting in R2019a

Starting in R2019a, specifying variant models in Model blocks is no longer supported. The Model Variants block, which was a Model block preconfigured to specify variant models, is also no longer supported. Loading a model that contains these blocks converts them to Variant Subsystem blocks that contain Model blocks that represent the variant choices.

Variant Subsystem blocks provide these advantages:

- You can mix Model blocks and Subsystem blocks as variant choices.
- You can specify variants that have different numbers of input and output ports.

The Variant Subsystem block parameter Analyze all choices during update diagram and generate preprocessor conditionals behaves differently from the Model block parameter Generate preprocessor conditionals. Instead of causing simulation and update diagram to compile the active variant only, the Variant Subsystem block parameter compiles all variants.
If you have scripts that use Model block parameters for variants, you must update them to use Variant Subsystem block parameters.

Extended Capabilities

**C/C++ Code Generation**
Generate C and C++ code using Simulink® Coder™.

Actual code generation support depends on block implementation.

**HDL Code Generation**
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic. For more information about using the Model block for HDL code generation and best practices and limitations, see “Model Referencing for HDL Code Generation” (HDL Coder).

See also “Generate Black Box Interface for Referenced Model” (HDL Coder) and “Generate Parameterized Code for Referenced Models” (HDL Coder).

**HDL Architecture**

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ModelReference (default)</td>
<td>When you want to generate code from a referenced model and any nested models, use the ModelReference implementation. For more information, see “How To Generate Code for a Referenced Model” (HDL Coder).</td>
</tr>
</tbody>
</table>
BlackBox

Use the BlackBox implementation to instantiate an HDL wrapper, or black box interface, for legacy or external HDL code. If you specify a black box interface, HDL Coder does not attempt to generate HDL code for the referenced model.

For more information, see “Generate Black Box Interface for Referenced Model” (HDL Coder).

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BlackBox</td>
<td>Use the BlackBox implementation to instantiate an HDL wrapper, or black box interface, for legacy or external HDL code. If you specify a black box interface, HDL Coder does not attempt to generate HDL code for the referenced model. For more information, see “Generate Black Box Interface for Referenced Model” (HDL Coder).</td>
</tr>
</tbody>
</table>

Black Box Interface Customization

For the BlackBox architecture, you can customize port names and set attributes of the external component interface. See “Customize Black Box or HDL Cosimulation Interface” (HDL Coder).

HDL Block Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BalanceDelays</td>
<td>Detects introduction of new delays along one path and inserts matching delays on the other paths. The default is inherit. See also “BalanceDelays” (HDL Coder).</td>
</tr>
<tr>
<td>ConstrainedOutputPipeline</td>
<td>Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. For more details, see “ConstrainedOutputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td>DistributedPipelining</td>
<td>Pipeline register distribution, or register retiming. The default is off. See also “DistributedPipelining” (HDL Coder).</td>
</tr>
<tr>
<td>DSPStyle</td>
<td>Synthesis attributes for multiplier mapping. The default is none. See also “DSPStyle” (HDL Coder).</td>
</tr>
<tr>
<td>InputPipeline</td>
<td>Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “InputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td>OutputPipeline</td>
<td>Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “OutputPipeline” (HDL Coder).</td>
</tr>
</tbody>
</table>
ReferenceModelPrefix

Prefix of the referenced model to insert in the generated code. The code generator applies this prefix to submodel file names and HDL identifiers. The default prefix is `modelname_` where `modelname` is the name of the referenced model.

**Note**

- If you specify an empty prefix, the code generator does not add a prefix to submodel file names. This can cause HDL compilation errors due to naming collisions between the models.
- If you use the referenced model as the DUT, the code generator ignores the prefix that you specify.

SharingFactor

Number of functionally equivalent resources to map to a single shared resource. The default is 0. See also “Resource Sharing” (HDL Coder).

StreamingFactor

Number of parallel data paths, or vectors, that are time multiplexed to transform into serial, scalar data paths. The default is 0, which implements fully parallel data paths. See also “Streaming” (HDL Coder).

**Fixed-Point Conversion**

Design and simulate fixed-point systems using Fixed-Point Designer™.

Actual data type support depends on block implementation.

**See Also**

`Simulink.SubSystem.convertToModelReference` | `Simulink.VariantManager.convertToVariant` | `depview` | `find_mdlrefs`

**Topics**

“Reference Existing Models”
“Parameterize Instances of a Reusable Referenced Model”
“Choose Simulation Modes for Model Hierarchies”
Introduced before R2006a
Model Info

Display model properties and text in model

Library: Simulink / Model-Wide Utilities
HDL Coder / Model-Wide Utilities

Description

The Model Info block displays model properties and text about a model on the mask of the block. Use the Model Info block dialog box to specify the content and format of the text that the block displays. You can select model properties to display on the block. In the text displayed on the block mask, Simulink replaces the property name with the current value of the property in the model.

Parameters

Specify Text and Properties to Display — Content and format of the text to display

no default

Use the Enter text and tokens to display on Model Info block edit box to specify the text and properties to display.

- In the edit box, enter any text you want to display on the block mask. Edit the default text Model Info.
- To display a model property on the block mask, select a property in the Model properties list and click the right arrow button.

The block adds a token of the form %<modelpropertypename> to the edit box. In the text the block mask displays, Simulink replaces the token with the value of the property.

For example, if you select Description in the Model properties list and click the right arrow button, then the token
%<Description>

appears in the right edit box.

2 You could add some explanatory text before the model property, e.g. “Model description:”.

3 When you click **Apply** or **OK**, Simulink displays your new text and the current value of the model property on the block mask in the Model Editor.

See “Version Information Properties” for descriptions of the model properties.

If you are interested in source control information, for a flexible interface to source control tools, use a project. See “Source Control in Projects”.

### Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>no</td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
</tr>
</tbody>
</table>

### Extended Capabilities

**C/C++ Code Generation**
Generate C and C++ code using Simulink® Coder™.

The Model Info block is ignored during code generation.
**HDL Code Generation**
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

**Best Practices**
When using Model Info blocks in models targeted for HDL code generation, consider using only ASCII characters in the text that you enter to display on the Model Info block. If you have non-ASCII characters in the generated HDL code, RTL simulation and synthesis tools can fail to compile the code.

**HDL Architecture**
This block has a single, default HDL architecture.

**HDL Block Properties**

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConstrainedOutputPipeline</td>
<td>Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. For more details, see “ConstrainedOutputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td>InputPipeline</td>
<td>Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “InputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td>OutputPipeline</td>
<td>Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “OutputPipeline” (HDL Coder).</td>
</tr>
</tbody>
</table>

**PLC Code Generation**
Generate Structured Text code using Simulink® PLC Coder™.

**See Also**
DocBlock
Topics
“Version Information Properties”
“Source Control in Projects”

Introduced before R2006a
Multiport Switch

Select output signal based on control signal

Library: Simulink / Signal Routing
        HDL Coder / Signal Routing

Description

The Multiport Switch block determines which of several inputs to the block passes to the output. The block bases this decision on the value of the first input. The first input is the control input and the remaining inputs are the data inputs. The value of the control input determines which data input passes to the output.

The table summarizes how the block interprets the control input and determines the data input that is passed to the output.

<table>
<thead>
<tr>
<th>Control Input</th>
<th>Truncation</th>
<th>Setting for Data Port Order</th>
<th>Block Behavior During Simulation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer value</td>
<td>None</td>
<td>Zero-based contiguous</td>
<td>Zero-based indexing</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>The control input is less than 0 or greater than the number of data inputs minus one.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>One-based contiguous</td>
<td>One-based indexing</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>The control input is less than 1 or greater than the number of data inputs.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Specify indices</td>
<td>Indices you specify</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>The control input does not correspond to any specified data port index.</td>
</tr>
</tbody>
</table>
Control Input | Truncation Setting for Data Port Order | Block Behavior During Simulation | Out-of-Range Condition
--- | --- | --- | ---
Not an integer value | The block truncates the value to an integer by rounding to zero. | Zero-based contiguous | Zero-based indexing | The truncated control input is less than 0 or greater than the number of data inputs minus one.
One-based contiguous | One-based indexing | The truncated control input is less than 1 or greater than the number of data inputs.
Specify indices | Indices you specify | The truncated control input does not correspond to any specified data port index.

For information on how the block handles the out-of-range condition, see “How the Block Handles an Out-of-Range Control Input” on page 1-1341.

**Multiport Switch Configured as an Index Vector Block**

An Index Vector is a special configuration of a Multiport Switch block in which you specify one data input and the control input is zero-based. The block output is the element of the input vector whose index matches the control input. For example, if the input vector is [18 15 17 10] and the control input is 3, the element that matches the index of 3 (zero-based) is 10, and that becomes the output value.

To configure a Multiport Switch block to work as an Index Vector block, set **Number of data ports** to 1 and **Data port order** to Zero-based contiguous.

**How the Block Handles an Out-of-Range Control Input**

For an input with an integer value less than intmax(‘int32’), the input is out of range when the value does not match any data port indices. For a control input that is not an integer value, the input is out of range when the truncated value does not match any data port indices. In both cases, the block behavior depends on your settings for **Data port for default case** and **Diagnostic for default case**.
**Note** If the control input is larger than `intmax('int32')`, the block wraps the input value to an integer.

### Behavior for Simulation

The following behavior applies only to simulation for your model.

<table>
<thead>
<tr>
<th>Data Port for Default Case</th>
<th>Diagnostic for Default Case</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Last data port</strong></td>
<td>None</td>
</tr>
<tr>
<td>Use the last data port and do not report any warning or error.</td>
<td>Use the last data port and report a warning.</td>
</tr>
<tr>
<td><strong>Additional data port</strong></td>
<td>Use the additional data port with a * label and do not report any warning or error.</td>
</tr>
</tbody>
</table>

### Behavior for Code Generation

The following behavior applies to code generation for your model.
Use Data Inputs That Have Different Dimensions

If two signals have a different number of dimensions or different dimension lengths, you can use the signals as data inputs to a Multiport Switch block. In the block dialog box, select the parameter **Allow different data input sizes**. In this case, the output of the block is a variable-size signal. If you do not select this parameter, the block generates an error.

For more information about the parameter, see “Allow different data input sizes (Results in variable-size output signal)” on page 1-0. For more information about variable-size signals, see “Variable-Size Signal Basics”. 

<table>
<thead>
<tr>
<th>Data Port for Default Case</th>
<th>Diagnostic for Default Case</th>
</tr>
</thead>
<tbody>
<tr>
<td>Last data port</td>
<td>None</td>
</tr>
<tr>
<td></td>
<td>Use the last data port.</td>
</tr>
<tr>
<td>Additional data port</td>
<td>Use the additional data port with a * label.</td>
</tr>
</tbody>
</table>
Rules That Determine the Block Behavior

You specify the number of data inputs with **Number of data ports**.

- If you set **Number of data ports** to 1, the block behaves as an *index selector* or *index vector* and not as a multiport switch. For more details, see “Multiport Switch Configured as an Index Vector Block” on page 1-1341.

- If you set **Number of data ports** to an integer greater than 1, the block behaves as a multiport switch. The block output is the data input that corresponds to the value of the control input. If at least one of the data inputs is a vector, the block output is a vector. In this case, the block expands any scalar inputs to vectors.

- If all the data inputs are scalar, the output is a scalar.

Guidelines on Setting Parameters for Enumerated Control Port

When the control port on the Multiport Switch block is of enumerated type, follow these guidelines:

<table>
<thead>
<tr>
<th>Scenario</th>
<th>What to Do</th>
<th>Rationale</th>
</tr>
</thead>
</table>
| The enumerated type contains a value that represents invalid, out-of-range, or uninitialized values. | • Set **Data port order** to Specify indices.  
• Set **Data port indices** to use this value for the last data port.  
• Set **Data port for default case** to Last data port. | This block configuration handles invalid values that the enumerated type explicitly represents. |
<p>| The enumerated type contains only valid enumerated values. However, a data input port can get invalid values of enumerated type. | • Set <strong>Data port for default case</strong> to Additional data port. | This block configuration handles invalid values that the enumerated type does not explicitly represent. |</p>
<table>
<thead>
<tr>
<th>Scenario</th>
<th>What to Do</th>
<th>Rationale</th>
</tr>
</thead>
</table>
| The enumerated type contains only valid enumerated values. Data input ports can never get invalid values of enumerated type. | • Set **Data port for default case** to **Last data port**.  
• Set **Diagnostic for default case** to **None**. | This block configuration avoids unnecessary diagnostic action. |
| The block does not have a data input port for every value of the enumerated type. | • Set **Data port for default case** to **Additional data port**. | This block configuration handles enumerated values that do not have a data input port, along with invalid values. |

**Limitations**

- If the data inputs to the Multiport Switch block are buses, the element names of both buses must be the same. Using the same element names ensures that the output bus has the same element names no matter which input bus the block selects. To ensure that your model meets this requirement, use a bus object to define the buses and set the **Element name mismatch** diagnostic to **error**. See “Connectivity Diagnostics Overview” for more information.

**Ports**

**Input**

**Port_1 — Control signal**

*scalar*

The control signal can be of any data type that Simulink supports, including fixed-point and enumerated types. When the control input is not an integer value, the block truncates the value to an integer by rounding to zero.

For information on control signals of enumerated type, see “Guidelines on Setting Parameters for Enumerated Control Port” on page 1-1344.
For information on how the block handles the out-of-range condition, see “How the Block Handles an Out-of-Range Control Input” on page 1-1341.

Limitations

- If the control signal is numeric, the control signal cannot be complex.
- If the control signal is an enumerated signal, the block uses the value of the underlying integer to select a data port.
- If the underlying integer does not correspond to a data port, an error occurs.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated

1 — First data input

scalar | vector | matrix | N-D array

First data input, specified as a scalar, vector, matrix, or N-D array. All input data signals can be of any data type that Simulink supports.

- If all the data inputs are scalar, the output is scalar
- If at least one of the data inputs is a vector, the block output is a vector. In this case, the block expands any scalar inputs to vectors.
- If any two nonscalar signals have a different number of dimensions or different dimension lengths, select the Allow different data input sizes check box. For more information, see “Use Data Inputs That Have Different Dimensions” on page 1-1343
- If any data signal is of an enumerated type, all others must be of the same enumerated type.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | string

2 — Second data input

scalar | vector | matrix | N-D array

Second data input, specified as a scalar, vector, matrix, or N-D array. All input data signals can be of any data type that Simulink supports.

- If all the data inputs are scalar, the output is scalar
- If at least one of the data inputs is a vector, the block output is a vector. In this case, the block expands any scalar inputs to vectors.
• If any two nonscalar signals have a different number of dimensions or different dimension lengths, select the *Allow different data input sizes* check box. For more information, see “Use Data Inputs That Have Different Dimensions” on page 1-1343.

• If any data signal is of an enumerated type, all others must be of the same enumerated type.

**Dependencies**

To enable this port, set **Number of data ports** to an integer greater than 1.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | string

**N — Nth data input**

scalar | vector | matrix | N-D array

Nth data input, specified as a scalar, vector, matrix, or N-D array. All input data signals can be of any data type that Simulink supports.

• If all the data inputs are scalar, the output is scalar
• If at least one of the data inputs is a vector, the block output is a vector. In this case, the block expands any scalar inputs to vectors.
• If any two nonscalar signals have a different number of dimensions or different dimension lengths, select the *Allow different data input sizes* check box. For more information, see “Use Data Inputs That Have Different Dimensions” on page 1-1343.

• If any data signal is of an enumerated type, all others must be of the same enumerated type.

**Dependencies**

To enable the Nth input port, set **Number of data ports** to an integer value greater than or equal to N.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | string

**— Data port for out-of-range inputs**

scalar | vector | matrix | N-D array

Input data port for out-of-range control signal inputs, specified as a scalar, vector, matrix, or N-D array. All input data signals can be of any data type that Simulink supports. If any data signal is of an enumerated type, all others must be of the same enumerated type. If
any two signals have a different number of dimensions or different dimension lengths, select the **Allow different data input sizes** check box. For more information, see “Use Data Inputs That Have Different Dimensions” on page 1-1343.

**Dependencies**

To create an additional data port for out-of-range control signal inputs, set **Data port for default case** to Additional data port. When you set **Data port for default case** to Last data port, the block uses the last data port for output when the control signal value does not match any data port indices.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus

**Output**

**Port_1 — Selected data input, based on control signal value**
scalar | vector | matrix | N-D array

The block outputs one of the data inputs, selected according to the control signal value. The output has the same dimensions as the corresponding data input. When you select the **Allow different data input sizes** check box, the output of the block is a variable size signal.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | string

**Parameters**

**Main**

**Data port order — Type of ordering for data input ports**
One-based contiguous | Zero-based contiguous | Specify indices

Specify the type of ordering for your data input ports.

- **Zero-based contiguous** — Block uses zero-based indexing for ordering contiguous data ports. This is the default value of the Index Vector block.
- **One-based contiguous** — Block uses one-based indexing for ordering contiguous data ports. This is the default value of the Multiport Switch block
• Specify indices — Block uses noncontiguous indexing for ordering data ports.

Tips

• When the control port is of enumerated type, select Specify indices.
• If you select Zero-based contiguous or One-based contiguous, verify that the control port is not of enumerated type. This configuration is deprecated and produces an error. You can run the Upgrade Advisor on your model to replace each Multiport Switch block of this configuration with a block that explicitly specifies data port indices. See “Model Upgrades”.
• Avoid situations where the block contains unused data ports for simulation or code generation. When the control port is of fixed-point or built-in data type, verify that all data port indices are representable with that type. Otherwise, the following block behavior occurs.

<table>
<thead>
<tr>
<th>If the Block Has Unused Data Ports and Data Port Order Is:</th>
<th>The Block Produces:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zero-based contiguous or One-based contiguous</td>
<td>A warning</td>
</tr>
<tr>
<td>Specify indices</td>
<td>An error</td>
</tr>
</tbody>
</table>

Dependencies

Selecting Zero-based contiguous or One-based contiguous enables the Number of data ports parameter.

Selecting Specify indices enables the Data port indices parameter.

Programmatic Use

Block Parameter: DataPortOrder

Type: character vector

Values: 'Zero-based contiguous' | 'One-based contiguous' | 'Specify indices'

Default: 'One-based contiguous' (Multiport Switch) 'Zero-based contiguous' (Index Vector)

Number of data ports — Number of data input ports
1 | 3 | integer between 1 and 65535
Specify the number of data input ports to the block. The total number of input ports is the number you specify, plus one for the control signal input port, and plus one more if you set **Data port for default case** to **Additional data port**.

**Dependencies**

To enable this parameter, set **Data port order** to **Zero-based contiguous** or **One-based contiguous**.

**Programmatic Use**

**Block Parameter:** Inputs  
**Type:** character vector  
**Values:** integer between 1 and 65536  
**Default:** '3' (Multiport Switch) '1' (Index Vector)

**Data port indices — Array of indices for data ports**

{1,2,3} (default) | array of indices

Specify an array of indices for your data ports. The block icon changes to match the data port indices you specify.

**Tips**

- To specify an array of indices that correspond to all values of an enumerated type, enter `enumeration('type_name')` for this parameter. Do not include braces.

  For example, `enumeration('MyColors')` is a valid entry.

- To enter specific values of an enumerated type, use the `type_name.enumerated_name` format. Do not enter the underlying integer value.

  For example, `{MyColors.Red, MyColors.Green, MyColors.Blue}` is a valid entry.

- To indicate that more than one value maps to a data port, use brackets.

  For example, the following entries are both valid:

  - `{[3,5],0,18}`

- If the control port is of fixed-point or built-in data type, the values for **Data port indices** must be representable with that type. Otherwise, an error appears at compile time to alert you to unused data ports.
• If the control port is of enumerated data type, the values for **Data port indices** must be enumerated values of that type.

• If **Data port indices** contains values of enumerated type, the control port must be of that data type.

**Dependencies**

To enable this parameter, set **Data port order** to **Specify indices**.

**Programmatic Use**

**Block Parameter:** DataPortIndices  
**Type:** character vector  
**Values:** array of indices  
**Default:** '{1,2,3}'

**Data port for default case — Port to use for out-of-range inputs**

*Last data port* (default) | *Additional data port*

Specify whether to use the last data port for out-of-range inputs, or to use an additional port. An asterisk (*) next to the port name indicates the port the block uses when the control port value does not match any data port indices.

• **Last data port** — Block uses the last data port for output when the control port value does not match any data port indices.

• **Additional data port** — Block uses an additional data port for output when the control port value does not match any data port indices.

**Tip**

If you set this parameter to **Additional data port** and **Number of data ports** is 3, the number of input ports on the block is 5. The first input is the control port, the next three inputs are data ports, and the fifth input is the default port for out-of-range inputs.

**Programmatic Use**

**Block Parameter:** DataPortForDefault  
**Type:** character vector  
**Values:** 'Last data port' | 'Additional data port'  
**Default:** 'Last data port'

**Diagnostic for default case — Diagnostic action when control port value does not match data port indices**

*Error* (default) | *Warning* | *None*
Specify the diagnostic action to take when the control port value does not match any data port indices.

- **None** — Produce no response.
- **Warning** — Display a warning and continue the simulation.
- **Error** — Terminate the simulation and display an error. In this case, the **Data port for default case** is used only for code generation and not simulation.

For more information, see “How the Block Handles an Out-of-Range Control Input” on page 1-1341.

**Programmatic Use**

**Block Parameter:** DiagnosticForDefault  
**Type:** character vector  
**Values:** 'None' | 'Warning' | 'Error'  
**Default:** 'Error'

**Sample time** — Specify sample time as a value other than -1

- -1 (default) | scalar | vector

Specify the sample time as a value other than -1. For more information, see “Specify Sample Time”.

**Dependencies**

This parameter is not visible unless it is explicitly set to a value other than -1. To learn more, see “Blocks for Which Sample Time Is Not Recommended”.

**Programmatic Use**

**Block Parameter:** SampleTime  
**Type:** character vector  
**Values:** scalar or vector  
**Default:** '-1'

**Signal Attributes**

**Require all data port inputs to have the same data type** — Require all inputs to have the same data type

- off (default) | on

Select this check box to require that all data input ports have the same data type. When you clear this check box, the block allows data port inputs to have different data types.
Programmatic Use

Block Parameter: InputSameDT
Type: character vector
Values: 'off' | 'on'
Default: 'off'

Output minimum — Minimum output value for range checking
[] (default) | scalar

Lower value of the output range that Simulink checks.

Simulink uses the minimum to perform:

- Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters”) for some blocks.
- Simulation range checking (see “Signal Ranges” and “Enable Simulation Range Checking”).
- Automatic scaling of fixed-point data types.
- Optimization of the code that you generate from the model. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. For more information, see “Optimize using the specified minimum and maximum values” (Simulink Coder).

**Note** Output minimum does not saturate or clip the actual output signal. Use the Saturation block instead.

Programmatic Use

Block Parameter: OutMin
Type: character vector
Values: '[]' | scalar
Default: '[]'

Output maximum — Maximum output value for range checking
[] (default) | scalar

Upper value of the output range that Simulink checks.

Simulink uses the maximum value to perform:
- Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters”) for some blocks.
- Simulation range checking (see “Signal Ranges” and “Enable Simulation Range Checking”).
- Automatic scaling of fixed-point data types.
- Optimization of the code that you generate from the model. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. For more information, see “Optimize using the specified minimum and maximum values” (Simulink Coder).

**Note** Output maximum does not saturate or clip the actual output signal. Use the Saturation block instead.

**Programmatic Use**

**Block Parameter:** OutMax  
**Type:** character vector  
**Values:** '[ ]' | scalar  
**Default:** '[ ]'

**Output data type — Specify the output data type**

Choose the data type for the output. The type can be inherited, specified directly, or expressed as a data type object such as Simulink.NumericType.

When you select an inherited option, the block behaves as follows:

- **Inherit: Inherit via internal rule**—Simulink chooses a data type to balance numerical accuracy, performance, and generated code size, while taking into account the properties of the embedded target hardware. If you change the embedded target settings, the data type selected by the internal rule might change. It is not always possible for the software to optimize code efficiency and numerical accuracy at the same time. If the internal rule doesn’t meet your specific needs for numerical accuracy or performance, use one of the following options:
• Specify the output data type explicitly.
• Explicitly specify a default data type such as \texttt{fixdt(1,32,16)} and then use the Fixed-Point Tool to propose data types for your model. For more information, see \texttt{fxptdlg}.
• To specify your own inheritance rule, use \texttt{Inherit: Inherit via back propagation} and then use a Data Type Propagation block. Examples of how to use this block are available in the Signal Attributes library Data Type Propagation Examples block.
• \texttt{Inherit: Inherit via back propagation} — Uses the data type of the driving block.

\textbf{Programmatic Use}  
\textbf{Block Parameter:} \texttt{OutDataTypeStr}  
\textbf{Type:} character vector  
\textbf{Values:} \texttt{'Inherit: Inherit via internal rule'} | \texttt{'Inherit: Inherit via back propagation'} | \texttt{'double'} | \texttt{'single'} | \texttt{'int8'} | \texttt{'uint8'} | \texttt{'int16'} | \texttt{'uint16'} | \texttt{'int32'} | \texttt{'uint32'} | \texttt{'int64'} | \texttt{'uint64'} | \texttt{'fixdt(1,16)'} | \texttt{'fixdt(1,16,0)'} | \texttt{'fixdt(1,16,2^0,0)'} | \texttt{'<data type expression>}'  
\textbf{Default:} \texttt{'Inherit: Inherit via internal rule'}

\textbf{Lock output data type setting against changes by the fixed-point tools — Prevent fixed-point tools from overriding data types}  
\texttt{off} (default) | \texttt{on}

Select to lock the output data type setting of this block against changes by the Fixed-Point Tool and the Fixed-Point Advisor. For more information, see “Use Lock Output Data Type Setting” (Fixed-Point Designer).

\textbf{Programmatic Use}  
\textbf{Block Parameter:} \texttt{LockScale}  
\textbf{Type:} character vector  
\textbf{Values:} \texttt{'off'} | \texttt{'on'}  
\textbf{Default:} \texttt{'off'}

\textbf{Integer rounding mode — Specify the rounding mode for fixed-point operations}  
\texttt{Floor} (default) | \texttt{Ceiling} | \texttt{Convergent} | \texttt{Nearest} | \texttt{Round} | \texttt{Simplest} | \texttt{Zero}

Choose one of these rounding modes.
Ceiling
Rounds both positive and negative numbers toward positive infinity. Equivalent to the MATLAB ceil function.

Convergent
Rounds number to the nearest representable value. If a tie occurs, rounds to the nearest even integer. Equivalent to the Fixed-Point Designer convergent function.

Floor
Rounds both positive and negative numbers toward negative infinity. Equivalent to the MATLAB floor function.

Nearest
Rounds number to the nearest representable value. If a tie occurs, rounds toward positive infinity. Equivalent to the Fixed-Point Designer nearest function.

Round
Rounds number to the nearest representable value. If a tie occurs, rounds positive numbers toward positive infinity and rounds negative numbers toward negative infinity. Equivalent to the Fixed-Point Designer round function.

Simplest
Automatically chooses between round toward floor and round toward zero to generate rounding code that is as efficient as possible.

Zero
Rounds number toward zero. Equivalent to the MATLAB fix function.

**Programmatic Use**

**Block Parameter:** RndMeth  
**Type:** character vector  
**Values:** 'Ceiling' | 'Convergent' | 'Floor' | 'Nearest' | 'Round' | 'Simplest' | 'Zero'  
**Default:** 'Floor'

**See Also**

For more information, see “Rounding” (Fixed-Point Designer).

**Saturate on integer overflow — Method of overflow action**

off (default) | on

Specify whether overflows saturate or wrap.
• **off** — Overflows wrap to the appropriate value that the data type can represent.
  
  For example, the number 130 does not fit in a signed 8-bit integer and wraps to -126.

• **on** — Overflows saturate to either the minimum or maximum value that the data type can represent.
  
  For example, an overflow associated with a signed 8-bit integer can saturate to -128 or 127.

**Tip**

• Consider selecting this check box when your model has a possible overflow and you want explicit saturation protection in the generated code.

• Consider clearing this check box when you want to optimize efficiency of your generated code.

Clearing this check box also helps you to avoid overspecifying how a block handles out-of-range signals. For more information, see “Check for Signal Range Errors”.

• When you select this check box, saturation applies to every internal operation on the block, not just the output or result.

• In general, the code generation process can detect when overflow is not possible. In this case, the code generator does not produce saturation code.

**Programmatic Use**

**Block Parameter:** SaturateOnIntegerOverflow

**Type:** character vector

**Values:** 'off' | 'on'

**Default:** 'off'

**Allow different data input sizes (Results in variable-size output signal) — Allow input signals with different sizes**

off (default) | on

Select this check box to allow input signals with different sizes.

• **on** — Allows input signals with different sizes, and propagate the input signal size to the output signal. In this mode, the block produces a variable-size output signal.
Off — Requires that all nonscalar data input signals be the same size.

Programmatic Use
Parameter: AllowDiffInputSizes
Type: character vector
Value: 'on' | 'off'
Default: 'off'

Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>bus</th>
<th>double</th>
<th>enumerated</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
<th>string</th>
</tr>
</thead>
</table>

| Direct Feedthrough | yes     |     |        |            |             |         |        |        |
| Multidimensional Signals | yes     |     |        |            |             |         |        |        |
| Variable-Size Signals   | yes     |     |        |            |             |         |        |        |
| Zero-Crossing Detection | no      |     |        |            |             |         |        |        |

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.
HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

<table>
<thead>
<tr>
<th>General</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CodingStyle</td>
<td>Specify whether to generate HDL code with case statements or if-else statements. By default, HDL Coder generates if-else statements. See also “CodingStyle” (HDL Coder).</td>
</tr>
<tr>
<td>ConstrainedOutputPipeline</td>
<td>Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. For more details, see “ConstrainedOutputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td>InputPipeline</td>
<td>Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “InputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td>OutputPipeline</td>
<td>Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “OutputPipeline” (HDL Coder).</td>
</tr>
</tbody>
</table>

Native Floating Point

| LatencyStrategy | Specify whether to map the blocks in your design to inherit, Max, Min, or Zero for the floating-point operator. The default is inherit. See also “LatencyStrategy” (HDL Coder). |

Complex Data Support

This block supports code generation for complex signals.

Using an Enumerated Type as Port Index

You can set Data port order to Specify indices, and enter enumeration values for the Data port indices. For example, you can connect the Enumerated Constant block to the Multiport Switch control port and use the enumerated types as data port indices.
PLC Code Generation
Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

See Also
Index Vector | Switch

Topics
“Variable-Size Signal Basics”

Introduced before R2006a
MultiStateImage

Display image reflecting input value

Library: Simulink / Dashboard

Description

The MultiStateImage block displays an image to indicate the value of the input signal. You can use the MultiStateImage block with other Dashboard blocks to build an interactive dashboard of controls and indicators for your model. You can specify pairs of input values and images to provide the information you want during simulation.

Connecting Dashboard Blocks

Dashboard blocks do not use ports to connect to signals. To connect Dashboard blocks to signals in your model, use connect mode. Connect mode facilitates the process of connecting Dashboard blocks in your model, especially when you want to connect multiple blocks at once. If you only want to connect a single Dashboard block, you can also use the Connection table in the block dialog box to make the connection.

Tip

To facilitate understanding and debugging your model, you can modify Dashboard block connections in your model during normal and accelerator mode simulations.

To enter connect mode, click the Connect button that appears above your unconnected Dashboard block when you pause on it.
In connect mode, when you select one or more signals or blocks, a list of parameters and signals available for connection appears. Select a signal or parameter from the list to connect the selected Dashboard block. To connect another Dashboard block, pause on the block and click the **Connect** button above it. Then, make a selection of signals and blocks in your model, and choose a signal or parameter to connect.
When you finish connecting the Dashboard blocks in your model, click the **Exit** button in the upper-right of the canvas to exit connect mode.

**Tip** You can hide the message shown on unconnected blocks using the `set_param` function and the `ShowInitialText` block parameter.

**Limitations**

- Dashboard blocks can only connect to real scalar signals.
- You cannot use the **Connection** table to connect a Dashboard block to a block that is commented out. When you connect a Dashboard block to a commented block using connect mode, the Dashboard block does not display the connected value until the you uncomment the block.
- Dashboard blocks cannot connect to signals inside referenced models.
- When you simulate a model hierarchy, Dashboard blocks inside referenced models do not update.
- Dashboard blocks do not support rapid accelerator simulation.
• Some signals are not available during accelerator mode simulations. To view such signals on a Dashboard block, mark the signal for logging.

**Parameters**

**Connection — Select a signal to connect and display**

Select the signal to connect using the Connection table. Populate the Connection table by selecting signals of interest in your model. Select the radio button next to the signal you want to display. Click **Apply** to connect the signal. To facilitate understanding and debugging your model, you can connect Dashboard blocks to signals in your model during simulation.

**Programmatic Use**

**Block Parameter:** Binding  
**Type:** Simulink.HMI.SignalSpecification  
**Default:** []

**Scale Mode — Specify how to scale image**

Specify how to scale the image.

**Fill with fixed Aspect Ratio** scales the image to the size of the block while retaining its original aspect ratio.

**Fixed** displays the image with its fixed true size.

**Fill** adjusts the image to fill the block.

**Programmatic Use**

**Block Parameter:** ScaleMode  
**Type:** string or character vector  
**Values:** 'Fill with fixed aspect ratio' | 'Fixed' | 'Fill'  
**Default:** 'Fill with fixed aspect ratio'

**States — Match values with images to display**

scalar and image
States match an image to display with a value of the connected variable or parameter. Each state consists of a **State** and an image, displayed as a **Thumbnail** in the **States** table for the MultiStateImage block.

- **State** — Connected variable or parameter value that causes the MultiStateImage block to display the corresponding image.
- **Thumbnail** — Image to display when the connected variable or parameter value matches the corresponding **State**.

Click the + button to add another state.

**Programmatic Use**

To programatically configure **State** and **Thumbnail** parameter values for the MultiStateImage block, use the **States** parameter. Specify the value for the **States** parameter as an array of structures with the fields:

- **State** — State value.
- **Size** — 1-by-2 vector containing the width and height for the image in that order, in pixels, specified as a **uint64**.
- **Image** — Character array of Base64 encoded image data.
- **Thumbnail** — Character array of Base64 encoded image data for the thumbnail image to display in the **States** table in the block dialog.

Include a structure in the array for each state you want to specify for the block. In the example, the `myBase64Image` and `myBase64Thumbnail` variables contain character vectors of the Base64 image data for the block image and thumbnail.

```matlab
state1.State = 1;
state1.Size = [uint64(400) uint64(400)];
state1.Image = myBase64Image;
state1.Thumbnail = myBase64Thumbnail;
state2.State = 2;
state2.Size = uint64(400) uint64(400)];
state1.Image = myBase64Image2;
state1.Thumbnail = myBase64Thumbnail2;
msiStates = [state1 state2];
```

**Block Parameter:** States  
**Type:** structure array
The block displays the default image when the connected signal value does not correspond to any of the specified states. Specify the DefaultImage parameter as a structure containing the fields:

- **Size** — 1-by-2 vector containing the width and height for the image in that order, in pixels, specified as uint64.
- **Image** — Character array of Base64 encoded image data.
- **Thumbnail** — Character array of Base64 encoded image data for the thumbnail image to display in the States table in the block dialog.

**Block Parameter:** DefaultImage  
**Type:** structure

**Label — Block label position**  
Top (default) | Bottom | Hide

Position of the block label. When the block is connected to an element in the model, the label is the name of the connected element.

**Programmatic Use**  
**Block Parameter:** LabelPosition  
**Type:** character vector  
**Values:** 'Top' | 'Bottom' | 'Hide'  
**Default:** 'Top'

**Block Characteristics**

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>double</th>
<th>enumerated</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

Dashboard blocks are ignored for code generation.

See Also
Lamp

Topics
"Tune and Visualize Your Model with Dashboard Blocks"
"Decide How to Visualize Simulation Data"

Introduced in R2016b
Mux

Combine input signals of same data type and numeric type into virtual vector

Library: Simulink / Commonly Used Blocks
       Simulink / Signal Routing
       HDL Coder / Commonly Used Blocks
       HDL Coder / Signal Routing

Description

The Mux block combines its inputs into a single vector output. An input can be a scalar or vector signal. All inputs must be of the same data type and numeric type. For information about creating and decomposing vectors, see “Mux Signals”.

The elements of the vector output signal take their order from the top to bottom, or left to right, input port signals. See “Port Location After Rotating or Flipping” for a description of the port order for various block orientations.

The nonvirtual components of a virtual signal are called regions. A virtual signal can contain the same region more than once. For example, if the same nonvirtual signal is connected to two input ports of a Mux block, the block outputs a virtual signal that has two regions. The regions behave as they would if they had originated in two different nonvirtual signals, even though the resulting behavior duplicates information.

Note Simulink provides several techniques for combining signals into a composite signal. For more information, see “Types of Composite Signals”.

Ports

Input

Port_1 — Accept nonbus vector signal to extract and output signals from real or complex values of any nonbus data type supported by Simulink

Port that accepts the input nonbus vector signal from which to extract and output signals.
Output

Port_1 — Output signals extracted from input vector signal
nonbus signal with real or complex values of any data type supported by Simulink

Output signals extracted from the input vector. The output signal ports are ordered from
top to bottom. See “Port Location After Rotating or Flipping” for a description of the port
order for various block orientations.

Parameters

Number of inputs — Inputs
2 (default) | scalar | vector | cell array | comma-separated list of signal names

Specify the number of input signals. You can also specify signal names and sizes. Use one
of these formats.

<table>
<thead>
<tr>
<th>Format</th>
<th>Block Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scalar</td>
<td>Specifies the number of inputs to the Mux block.</td>
</tr>
<tr>
<td></td>
<td>When you use this format, the block accepts scalar or vector signals of any size. Simulink assigns each input the name signalN, where N is the input port number.</td>
</tr>
<tr>
<td>Vector</td>
<td>The length of the vector specifies the number of inputs. Each element specifies the size of the corresponding input.</td>
</tr>
<tr>
<td></td>
<td>A positive value specifies that the corresponding port can accept only vectors of that size. For example, [2 3] specifies two input ports of sizes 2 and 3, respectively. If an input signal width does not match the expected width, an error message appears. A value of -1 specifies that the corresponding port can accept scalars or vectors of any size.</td>
</tr>
<tr>
<td>Format</td>
<td>Block Behavior</td>
</tr>
<tr>
<td>-------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Cell array</td>
<td>The length of the cell array specifies the number of inputs. The value of each cell specifies the size of the corresponding input. A scalar value N specifies a vector of size N. A value of -1 means that the corresponding port can accept scalar or vector signals of any size.</td>
</tr>
<tr>
<td>Signal name list</td>
<td>You can enter a list of signal names separated by commas. Simulink assigns each name to the corresponding port and signal. For example, if you enter position, velocity, the Mux block has two inputs, named position and velocity.</td>
</tr>
</tbody>
</table>

**Tip** If you specify a scalar for the **Number of inputs** parameter and all of the input ports are connected, as you draw a new signal line close to input side of a Mux block, Simulink adds a port and updates the parameter.

**Programmatic Use**

**Block Parameter:** Inputs

- **Type:** scalar, vector, cell array, signal name list
- **Values:** number, vector of port numbers, cell array, or list of signal names
- **Default:** {'2'}

**Display option — Displayed block icon**

- **bar** (default) | **nonesignal**

By default, the block icon is a solid bar of the block foreground color. To display the icon as a hollow bar containing input signal names, select **signals**. To display the icon as a box containing the block type name, select **none**.

**Programmatic Use**

**Block Parameter:** DisplayOption

- **Type:** character vector
- **Values:** 'bar' 'signals' 'none''bar'
- **Default:** 'bar'
Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>double</th>
<th>enumerated</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

Actual data type or capability support depends on block implementation.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture
This block has a single, default HDL architecture.
**HDL Block Properties**

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
<th>Default</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConstrainedOutputPipeline</td>
<td>Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. For more details, see “ConstrainedOutputPipeline” (HDL Coder).</td>
<td></td>
<td></td>
</tr>
<tr>
<td>InputPipeline</td>
<td>Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “InputPipeline” (HDL Coder).</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OutputPipeline</td>
<td>Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “OutputPipeline” (HDL Coder).</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Complex Data Support**

This block supports code generation for complex signals.

**Restrictions**

Buses are not supported for HDL code generation.

**PLC Code Generation**

Generate Structured Text code using Simulink® PLC Coder™.

**Fixed-Point Conversion**

Design and simulate fixed-point systems using Fixed-Point Designer™.

Actual data type or capability support depends on block implementation.

**See Also**

Bus Creator | Bus to Vector | Demux | Vector Concatenate

**Topics**

“Virtual Signals”
“Types of Composite Signals”
“Simplify Bus Interfaces”

Introduced before R2006a
**Out Bus Element, Bus Element Out**

Specify signal that connects to output port

**Library:**
- Simulink / Ports & Subsystems
- Simulink / Sinks
- HDL Coder / Ports & Subsystems
- HDL Coder / Sinks

**Description**

The Out Bus Element block, also known as the Bus Element Out block, combines the functionality of an Outport block and a Bus Creator block. This block is of the Outport block type and all Out Bus Element blocks that use the same port share a Block Parameters dialog box. In the Block Parameters dialog box, you can reorder bus elements by dragging and dropping a signal in the list of signals.

To create a virtual bus output, you can use an Out Bus Element block for each signal that you want the bus to contain. In the Block Parameters dialog box, you can select the **Add a new signal** or **Add a new sub-bus** buttons to create related Out Bus Element blocks.

When you save output data to the workspace or a file, bus data defined by groups of root-level Out Bus Element blocks gets logged along with root-level Outport block data.

**Specify a Signal**

To change the signal associated with a bus element port block, you must edit the label next to the block by double-clicking the text. Dot notation indicates signal hierarchy within the bus. To reduce the size of the label, you can display only the signal name by selecting **Compact Notation** from the ellipsis menu that appears when you select the block. To show the signal hierarchy again, you can select **Expanded Notation** from the menu.
Ports

Input

Port_1 — Output bus element signal

Signal to be included in an output bus that is associated with an external subsystem or model port.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus

Parameters

Port name — Name of output port

0utBus (default) | port name

Specify a port name that is not already in use by another block or port. The name appears on the parent Subsystem or Model block. The name also appears next to the block. Multiple blocks can access the same port.

Programmatic Use

Block Parameter: PortName

Type: character vector

Values: 'OutBus' | '<port name>'

Default: 'OutBus'

Port number — Position of port on parent block

1 (default) | real integer

Specify the order in which the port that corresponds to the block appears on the parent Subsystem or Model block.

• If you add a block that creates another port, the port number is the next available number.
• Deleting all blocks associated with a port deletes the port. Other ports are renumbered so that they are sequential and do not skip any numbers.
• Specifying a port number that exceeds the number of ports creates a port for that number and for any skipped sequential numbers.
**Programmatic Use**

**Block Parameter:** Port  
**Type:** character vector  
**Values:** real integer  
**Default:** '1'

**Set color — Specify block background color**  
Black (default) | White | Red | Green | Blue | ...

Specify the background color for bus element port blocks. This specification sets the color of blocks associated with selected bus elements, or of all blocks associated with the port if you do not select any bus elements.

You can specify Black, White, Red, Green, Blue, Cyan, Magenta, Yellow, Gray, Light Blue, Orange, Dark Green, and More Colors.

**Programmatic Use**

**Block Parameter:** BackgroundColor  
**Type:** character vector  
**Values:** 'black' | 'white' | 'red' | 'green' | 'blue' | 'cyan' | 'magenta' | 'yellow' | 'gray' | 'lightBlue' | 'orange' | 'darkGreen' | '[r,g,b]' where r, g, and b, are the red, green, blue values of the color in the range 0.0 to 1.0  
**Default:** 'black'

**Filter — Filter set of displayed signals**  
**text**

Specify a search term to use for filtering a long list of bus elements. Do not enclose the search term in quotation marks. The filter does a partial string search and supports regular expressions. To use a regular expression character as a literal, include an escape character (\). For example, to use a question mark, type sig\?1.

**Signal Attributes**

To specify signal attributes, click the pencil button or the specified attributes next to a signal name.

**Data Type — Data type**  
**Inherit:** auto (default) | double | single | int8 | uint8 | int16 | uint16 | int32 | uint32 | int64 | uint64 | fixdt(1,16) | fixdt(1,16,0) | fixdt(1,16,2^0,0) | Enum: <class name> | Bus: <object name> | <data type expression>
Specify the data type. The type can be inherited, specified directly, or expressed as a data type object such as Simulink.NumericType. Do not specify a bus object as the expression.

**Dimensions — Signal dimensions**
-1 (default) | integer | [integer, integer]

Specify the dimensions of a signal.

<table>
<thead>
<tr>
<th></th>
<th>The signal can have any dimensions.</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>The signal must be a vector of size N.</td>
</tr>
<tr>
<td>[R C]</td>
<td>The signal must be a matrix having R rows and C columns.</td>
</tr>
</tbody>
</table>

**Programmatic Use**
**Block Parameter:** PortDimensions
**Type:** character vector  
**Values:** ' -1' | integer | [integer, integer]  
**Default:** ' -1'

**Dims Mode — Allow variable-size signals**
Inherit (default) | Fixed | Variable

Specify the type of signals allowed. To allow variable-size and fixed-size signals, select Inherit. To allow only variable-size signals, select Variable. To not allow variable-size signals, select Fixed.

**Dependencies**

When the signal is variable-sized, the Dimensions parameter specifies the maximum dimensions of the signal.

**Command-Line Information**
**Parameter:** VarSizeSig
**Type:** character vector  
**Value:** 'Inherit' | 'No' | 'Yes'  
**Default:** 'Inherit'

**Unit — Physical unit of the signal**
inheritor (default) | <Enter unit>

Specify the physical unit of the signal. For a list of supported units, see Allowed Unit Systems.
**Programmatic Use**

**Block Parameter**: Unit  
**Type**: character vector  
**Values**: 'inherit' | '<Enter unit>'  
**Default**: 'inherit'

**Sample time — Specify sample time**  
-1 (default) | scalar

Specify the discrete interval between sample time hits or specify another appropriate sample time such as continuous or inherited.

By default, the signal inherits its sample time. To set a different sample time, enter a valid sample time based on the table in “Types of Sample Time”.

**Programmatic Use**

**Block Parameter**: SampleTime  
**Type**: character vector  
**Values**: scalar  
**Default**: '-1'

**Complexity — Signal type**  
auto (default) | real | complex

Specify the numeric type of the signal. To choose the numeric type of the signal, select auto. Otherwise, choose a real or complex signal type.

**Programmatic Use**

**Block Parameter**: SignalType  
**Type**: character vector  
**Values**: 'auto' | 'real' | 'complex'  
**Default**: 'auto'

**Minimum — Minimum value**  
[] (default) | scalar

Lower value of the range that Simulink checks.

This number must be a finite real double scalar value.

Simulink software uses this value to perform:

- Simulation range checking (see “Signal Ranges”).
• Automatic scaling of fixed-point data types.
• Optimization of the code that you generate from the model. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. For more information, see “Optimize using the specified minimum and maximum values” (Simulink Coder).

**Programmatic Use**

**Block Parameter**: OutMin  
**Type**: character vector  
**Values**: '[]'| scalar  
**Default**: '[]'

**Maximum — Maximum value**

[ ] (default) | scalar

Upper value of the range that Simulink checks.

This number must be a finite real double scalar value.

Simulink software uses this value to perform:

• Simulation range checking (see “Signal Ranges”).
• Automatic scaling of fixed-point data types.
• Optimization of the code that you generate from the model. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. For more information, see “Optimize using the specified minimum and maximum values” (Simulink Coder).

**Programmatic Use**

**Block Parameter**: OutMax  
**Type**: character vector  
**Values**: '[]'| scalar  
**Default**: '[]'

**Block Characteristics**

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>bus</th>
<th>double</th>
<th>enumerated</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
<th>string</th>
</tr>
</thead>
</table>

1-1379
### Extended Capabilities

#### C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

#### HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

This block can be used to simplify the Subsystem bus interfaces when you use the block in subsystems that generate HDL code, but is not included in the hardware implementation.

To learn more about using buses for HDL code generation, see “Buses” (HDL Coder) and “Use Bus Signals to Improve Readability of Model and Generate HDL Code” (HDL Coder).

#### Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

### See Also

**Blocks**
Bus Creator | In Bus Element | Outport
Topics
“Simplify Bus Interfaces”
“Types of Composite Signals”
“Getting Started with Buses”

Introduced in R2017a
Outport

Create output port for subsystem or external output

Library:  
- Simulink / Commonly Used Blocks
- Simulink / Ports & Subsystems
- Simulink / Sinks
- HDL Coder / Commonly Used Blocks
- HDL Coder / Ports & Subsystems
- HDL Coder / Sinks

Description

Outport blocks link signals from a system to a destination outside of the system. They can connect signals flowing from a subsystem to other parts of the model. They can also supply external outputs at the top level of a model hierarchy.

Simulink software assigns Outport block port numbers according to these rules:

- It automatically numbers the Outport blocks within a root-level system or subsystem sequentially, starting with 1.
- If you add an Outport block, it is assigned the next available number.
- If you delete an Outport block, other port numbers are automatically renumbered to ensure that the Outport blocks are in sequence and that no numbers are omitted.

Outport Blocks in a Subsystem

Outport blocks in a subsystem represent outputs from the subsystem. A signal arriving at an Outport block in a subsystem flows out of the associated output port on that Subsystem block. The Outport block associated with an output port is the block whose Port number parameter matches the relative position of the output port on the Subsystem block. For example, the Outport block whose Port number parameter is 1 sends its signal to the block connected to the topmost output port on the Subsystem block.

If you renumber the Port number of an Outport block, the block becomes connected to a different output port. The block continues to send the signal to the same block outside the subsystem.
Tip For models that include bus signals composed of many bus elements, consider using In Bus Element and Out Bus Element blocks. These blocks:

- Reduce signal line complexity and clutter in a block diagram.
- Make it easier to change the interface incrementally.
- Allow access to a bus element closer to the point of usage, avoiding the use of a Bus Selector and Goto block configuration.

The In Bus Element block is of block type Inport and the Out Bus Element block is of block type Outport.

Top-Level Outport Block in a Model Hierarchy

Outport blocks at the top level of a model hierarchy have two uses. They can supply external outputs to the base MATLAB workspace, and they provide a means for the linmod and trim analysis functions to obtain output from the system.

To supply external outputs to the workspace, use the Configuration Parameters > Data Import/Export pane (see Exporting Output Data to the MATLAB Workspace) or the sim command. For example, if a system has more than one Outport block and the save format is array, the following command:

\[ [t,x,y] = \text{sim}(\ldots); \]

writes \( y \) as a matrix, with each column containing data for a different Outport block. The column order matches the order of the port numbers for the Outport blocks.

If you specify more than one variable name after the second (state) argument, data from each Outport block is written to a different variable. For example, if the system has two Outport blocks, to save data from Outport block 1 to speed and the data from Outport block 2 to dist, specify this command:

\[ [t,x,speed,dist] = \text{sim}(\ldots); \]

Connecting Buses to Root-Level Outports

A root-level Outport block in a model can accept a virtual bus only if all elements of the bus have the same data type. The Outport block automatically unifies the bus to a vector having the same number of elements as the bus, and provides that vector as output.
If you want a root-level Outport of a model to accept a bus signal that contains mixed types, set Outport block **Data type** to **Bus: <object name>**. If the bus signal is virtual, it is converted to nonvirtual, as described in “Bus Conversion”.

**Associate Root-Level Outport Block with Simulink.Signal Object**

To associate a root-level Outport block with a Simulink.Signal object, use the Model Data Editor. See “For Signals”.

**Ports**

**Input**

**Port_1 — Outport signal**

*scalar | vector*

Input signal that flows through the outport to an external subsystem or model.

An Outport block can accept fixed-point and enumerated data types when the block is not a root-level output port. The complexity and data type of the block output are the same as its input.

*Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus*

**Parameters**

**Main**

**Port number — Position of port on parent block**

*1 (default) | real integer*

Specify the order in which the port that corresponds to the block appears on the parent Subsystem or Model block.
• If you add a block that creates another port, the port number is the next available number.
• Deleting all blocks associated with a port deletes the port. Other ports are renumbered so that they are sequential and do not skip any numbers.
• Specifying a port number that exceeds the number of ports creates a port for that number and for any skipped sequential numbers.

**Programmatic Use**
**Block Parameter:** Port
**Type:** character vector
**Values:** real integer
**Default:** '1'

**Signal name — Signal name**
' ' (default) | character vector

Specify the name of the corresponding signal data in the generated code. Use this parameter to specify a name for the signal data when you apply a storage class to a root-level Outport block.

**Programmatic Use**
**Block Parameter:** SignalName
**Type:** character vector
**Values:** character vector
**Default:** ' '

**Icon display — Icon display**
Port number (default) | Signal name | Port number and signal name

Specify the information displayed on the block icon.

**Programmatic Use**
**Block Parameter:** IconDisplay
**Type:** character vector
**Values:** 'Signal name' | 'Port number' | 'Port number and signal name'
**Default:** 'Port number'

**Specify output when source is unconnected — Specify unconnected output value**
off (default) | on

Specify a constant output value for when source is not connected.
Dependency
Available for unconnected Outport blocks inside a Variant Subsystem block.

Programmatic Use
Block Parameter: OutputWhenUnconnected
Type: character vector
Values: 'on' | 'off'
Default: 'off'

Constant value — Output value when the block is not connected
0 (default) | scalar | vector
Specify a constant output value for when source is not connected.

Dependency
Available in a Variant Subsystem block for an unconnected Outport block with Specify output when source is unconnected selected.

Programmatic Use
Block Parameter: OutputWhenUnconnectedValue
Type: character vector
Values: scalar | vector
Default: '0'

Interpret vector parameters as 1-D — Treat vectors as 1-D
off (default) | on
Select this check box to create a vector of length N at the output if the Constant value parameter evaluates to an N-element row or column vector.

Dependency
Available in a Variant Subsystem block on an unconnected Outport block when you select the Specify output when source is unconnected parameter.

Programmatic Use
Block Parameter: VectorParams1DForOutWhenUnconnected
Type: character vector
Values: 'on' | 'off'
Default: 'off'

Ensure outport is virtual — Ensure that outport is virtual
off (default) | on
Clear this check box to specify that Simulink uses a signal buffer on output port. This buffer ensures consistent initialization of the Outport block signal.

If you select this check box, Simulink tries to remove the signal buffer.

- If the signal buffer is not needed, Simulink removes the buffer.
- If the signal buffer is needed for data consistency and proper execution, Simulink displays an error indicating the buffer could not be removed.

Allow partial writes through Assignment blocks.

For examples with conditional writes and partial writes, see “Ensure Outport is Virtual”.

**Dependency**

This parameter applies to these blocks:

- Conditional subsystem
- Assignment
- Merge
- Model with root Outport block

**Programmatic Use**

**Block Parameter:** EnsureOutportIsVirtual  
**Type:** character vector  
** Values:** 'on' | 'off'  
**Default:** 'off'

**Source of initial output value — Source of initial output value**

Dialog (default) | Input signal

Select the source of the initial output value of the block. Select Dialog to specify that the initial output value is the value of the Initial output parameter. Select Input signal to specify that the initial output value is inherited from the input signal. See “Conditional Subsystem Initial Output Values”.

**Tips**

- If you are using classic initialization mode, selecting Input signal causes an error. To inherit the initial output value from the input signal, set this parameter to Dialog and specify [] (empty matrix) for the Initial output value. For more information, see “Conditional Subsystem Initial Output Values”.
Dependencies

Available for Outport blocks inside a conditional subsystem.

Selecting Dialog exposes the following parameters:

- **Output when disabled**
- **Initial output**

Programmatic Use

Block Parameter: SourceOfInitialOutputValue
Type: character vector
Values: 'Dialog' | 'Input signal'
Default: 'Dialog'

Output when disabled — Output when disabled

*held* (default) | *reset*

Specify what happens to the block output when the subsystem is disabled. Select *held* to indicate that the output is held when the subsystem is disabled. Select *reset* to indicate that the output is reset to the value given by **Initial output** when the subsystem is disabled.

Dependencies

Available when you select Dialog as the value for the Source of initial output parameter for an Outport block inside a conditional subsystem with valid enabling and disabling semantics. For example, the parameter is available for an Outport block inside an Enabled Subsystem block and not for an Outport block inside a Triggered Subsystem block.

If an Outport block is inside a Function-Call Subsystem block, this parameter is only meaningful if the Function-Call Subsystem block is connected to a state in a Stateflow chart. For more information, see “Bind a Function-Call Subsystem to a State” (Stateflow).

When connecting the output of a conditional subsystem to a Merge block, set this parameter to held. Setting it to reset returns an error.

Programmatic Use

Block Parameter: OutputWhenDisabled
Type: character vector
Values: 'held' | 'reset'
**Default:** 'held'

**Initial output — Initial output for conditionally executed subsystems**

[ ] (default) | scalar | structure

For conditionally executed subsystems, specify the block output before the subsystem executes and while it is disabled. Specify [ ] to inherit the initial output value from the input signal. For more information, see “Conditional Subsystem Initial Output Values”.

For information about specifying an initial condition structure, see “Specify Initial Conditions for Bus Signals”.

**Tips**

If the conditional subsystem is driving a Merge block, you do not need to specify an Initial Condition (IC) for the Outport block. For more information, see “Underspecified initialization detection”.

**Dependencies**

Available when **Source of initial output value** is set to Dialog for an Outport block in a conditional subsystem.

**Limitations**

- This block does not allow an initial output of inf or NaN.
- When the input is a virtual bus, an Initial output value [ ] is treated as double(0).
- When the input contains a nonvirtual bus, Initial output does not support nonzero scalar values.

**Programmatic Use**

**Block Parameter:** InitialOutput  
**Type:** character vector  
**Values:** ' [ ] ' | scalar | structure  
**Default:** ' [ ] '

**Signal Attributes**

Click the **Show data type assistant** button to display the **Data Type Assistant**, which helps you set the data type attributes. For more information, see “Specify Data Types Using Data Type Assistant”.

1-1389
**Minimum — Minimum output value**

[] (default) | scalar

Lower value of the output range that Simulink checks.

This number must be a finite real double scalar value.

**Note** If you specify a bus object as the data type for this block, do not set the minimum value for bus data on the block. Simulink ignores this setting. Instead, set the minimum values for bus elements of the bus object specified as the data type. For information on the Minimum property of a bus element, see `Simulink.BusElement`.

Simulink software uses this value to perform:

- Simulation range checking (see “Signal Ranges”).
- Automatic scaling of fixed-point data types.
- Optimization of the code that you generate from the model. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. For more information, see “Optimize using the specified minimum and maximum values” (Simulink Coder).

**Programmatic Use**

**Block Parameter:** OutMin

**Type:** character vector

**Values:** '[]' | scalar

**Default:** '[]'

**Maximum — Maximum output value**

[] (default) | scalar

Upper value of the output range that Simulink checks.

This number must be a finite real double scalar value.

**Note** If you specify a bus object as the data type for this block, do not set the maximum value for bus data on the block. Simulink ignores this setting. Instead, set the maximum values for bus elements of the bus object specified as the data type. For information on the Maximum property of a bus element, see `Simulink.BusElement`. 
Simulink software uses this value to perform:

• Simulation range checking (see “Signal Ranges”).
• Automatic scaling of fixed-point data types.
• Optimization of the code that you generate from the model. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. For more information, see “Optimize using the specified minimum and maximum values” (Simulink Coder).

**Programmatic Use**

**Block Parameter**: OutMax

**Type**: character vector

**Values**: ' [ ] ' | scalar

**Default**: ' [ ] '

**Data type — Output data type**

Inherit: auto (default) | double | single | int8 | uint8 | int16 | uint16 | int32 | uint32 | int64 | uint64 | fixdt(1,16) | fixdt(1,16,0) | fixdt(1,16,2^0,0) | Enum: <class name> | Bus: <object name> | <data type expression>

Specify the output data type of the external input. The type can be inherited, specified directly, or expressed as a data type object such as Simulink.NumericType.

**Tips**

You cannot enter the name of a Simulink.Bus object as a data type expression. To specify the **Data type** for the block using a Bus object, select the Bus: <object name> option and replace <object name> with the name of the Bus object.

**Lock output data type setting against changes by the fixed-point tools — Prevent fixed-point tools from overriding data types**

off (default) | on

Select to lock the output data type setting of this block against changes by the Fixed-Point Tool and the Fixed-Point Advisor. For more information, see “Use Lock Output Data Type Setting” (Fixed-Point Designer).

**Programmatic Use**

**Block Parameter**: LockScale

**Type**: character vector

**Values**: ' off ' | ' on '
Default: 'off'

Output as nonvirtual bus in parent model — Outport as nonvirtual bus in parent model
off (default) | on

Specify the outport bus to be nonvirtual in the parent model. Select this parameter if you want the bus emerging in the parent model to be nonvirtual. The bus that is input to the port can be virtual or nonvirtual, regardless of the setting of Output as nonvirtual bus in parent model.

Clear this parameter if you want the bus emerging in the parent model to be virtual.

Tips

- In a nonvirtual bus, all signals must have the same sample time, even if the elements of the associated bus object specify inherited sample times. Any operation that would result in a nonvirtual bus that violates this requirement generates an error. For details, see “Multirate Bus Input for Referenced Models”.

To use a multirate signal for a virtual bus, in the root-level Outport block, set the Sample time parameter to inherited (-1).

- For the top model in a model reference hierarchy, code generation creates a C structure to represent the bus signal output of this block.

- For referenced models, select this option to create a C structure. Otherwise, code generation creates an argument for each leaf element of the bus.

Dependency

Available when the Data type is set to Bus: <object name>.

Programmatic Use

Block Parameter: BusOutputAsStruct
Type: character vector
Values: 'off' | 'on'
Default: 'off'

Unit (e.g., m, m/s^2, N*m) — Physical unit of the input signal to the block
inherit (default) | <Enter unit>

Specify the physical unit of the input signal to the block. To specify a unit, begin typing in the text box. As you type, the parameter displays potential matching units. For a list of supported units, see Allowed Unit Systems.
To constrain the unit system, click the link to the right of the parameter:

- If a Unit System Configuration block exists in the component, its dialog box opens. Use that dialog box to specify allowed and disallowed unit systems for the component.
- If a Unit System Configuration block does not exist in the component, the model Configuration Parameters dialog box displays. Use that dialog box to specify allowed and disallowed unit systems for the model.

**Programmatic Use**
**Block Parameter:** Unit  
**Type:** character vector  
**Values:** 'inherit' | '<Enter unit>'  
**Default:** 'inherit'

**Port dimensions (-1 for inherited) — Port dimensions**  
-1 (default) | integer | [integer, integer]

Specify the dimensions that a signal must have to be connected to this Outport block.

<table>
<thead>
<tr>
<th>-1</th>
<th>A signal of any dimensions can be connected to this port.</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>The signal connected to this port must be a vector of size N.</td>
</tr>
<tr>
<td>[R C]</td>
<td>The signal connected to this port must be a matrix having R rows and C columns.</td>
</tr>
</tbody>
</table>

**Programmatic Use**
**Block Parameter:** PortDimensions  
**Type:** character vector  
**Values:** '-1' | integer | [integer, integer]  
**Default:** '-1'

**Variable-size signal — Allow variable-size signals**  
Inherit (default) | No | Yes

Specify the type of signals allowed at the output of this port. To allow variable-size and fixed-size signals, select Inherit. To allow only variable-size signals, select Yes. To allow only fixed-size signals, select No.

**Dependencies**

When the signal at this port is a variable-size signal, the **Port dimensions** parameter specifies the maximum dimensions of the signal.
**Command-Line Information**

**Parameter:** VarSizeSig  
**Type:** character vector  
**Value:** 'Inherit' | 'No' | 'Yes'  
**Default:** 'Inherit'

**Sample time (-1 for inherited) — Specify sample time**  
-1 (default) | scalar

Specify the discrete interval between sample time hits or specify another appropriate sample time such as continuous or inherited.

By default, the block inherits its sample time based upon the context of the block within the model. To set a different sample time, enter a valid sample time based upon the table in “Types of Sample Time”.

**Programmatic Use**

**Block Parameter:** SampleTime  
**Type:** character vector  
**Values:** scalar  
**Default:** '-1'

**Signal type — Output signal type**

auto (default) | real | complex

Specify the numeric type of the signal output. To choose the numeric type of the signal that is connected to its input, select auto. Otherwise, choose a real or complex signal type.

**Programmatic Use**

**Block Parameter:** SignalType  
**Type:** character vector  
**Values:** 'auto' | 'real' | 'complex'  
**Default:** 'auto'

**Block Characteristics**

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>bus</th>
<th>double</th>
<th>enumerated</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
<th>string</th>
</tr>
</thead>
</table>

1-1394
### Extended Capabilities

#### C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

#### HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

#### HDL Architecture
This block has a single, default HDL architecture.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>yes</td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>yes</td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
</tr>
</tbody>
</table>
## HDL Block Properties

<table>
<thead>
<tr>
<th>BidirectionalPort Setting</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>on</td>
<td>Specify the port as bidirectional. The following requirements apply:</td>
</tr>
<tr>
<td></td>
<td>• The port must be in a Subsystem block with black box implementation.</td>
</tr>
<tr>
<td></td>
<td>• There must also be no logic between the bidirectional port and the</td>
</tr>
<tr>
<td></td>
<td>corresponding top-level DUT subsystem port.</td>
</tr>
<tr>
<td></td>
<td>For more information, see “Specify Bidirectional Ports” (HDL Coder).</td>
</tr>
<tr>
<td>off (default)</td>
<td>Do not specify the port as bidirectional.</td>
</tr>
<tr>
<td><strong>Target Specification</strong></td>
<td><strong>IOInterface</strong></td>
</tr>
<tr>
<td>-------------------------</td>
<td>----------------</td>
</tr>
<tr>
<td></td>
<td>Target platform interface type for DUT ports, specified as a character vector. The IOInterface block property is ignored for Inport and Outport blocks that are not DUT ports.</td>
</tr>
</tbody>
</table>

To specify valid IOInterface settings, use the HDL Workflow Advisor:

1. In the HDL Workflow Advisor, in the **Set Target > Set Target Interface** step, in the **Target platform interface table**, in the **Target Platform Interfaces** column, use the drop-down list to set the target platform interface type.

2. Save the model.

The IOInterface value is saved as an HDL block property of the port.

For example, to view the IOInterface value, if the full path to your DUT port is `hdlcoder_led_blinking/led_counter/LED`, enter:

    hdlget_param('hdlcoder_led_blinking/led_counter/LED', 'IOInterface')
**Target Specification**

| IOInterfaceMapping | Target platform interface port mapping for DUT ports, specified as a character vector. The IOInterfaceMapping block property is ignored for Inport and Outport blocks that are not DUT ports.

To specify valid IOInterfaceMapping settings, use the HDL Workflow Advisor:

1. In the HDL Workflow Advisor, in the **Set Target > Set Target Interface** step, in the **Target platform interface table**, in the **Target Platform Interfaces** column, use the drop-down list to set the target platform interface type.

2. In the **Bit Range / Address / FPGA Pin** column, if you want to change the default value, enter a target platform interface mapping.

3. Save the model.

   The IOInterfaceMapping value is saved as an HDL block property of the port.

   For example, to view the IOInterfaceMapping value, if the full path to your DUT port is `hdlcoder_led_blinking/led_counter/LED`, enter:

   ```matlab
   hdlget_param('hdlcoder_led_blinking/led_counter/LED',... 'IOInterfaceMapping')
   ```

**PLC Code Generation**

Generate Structured Text code using Simulink® PLC Coder™.

**Fixed-Point Conversion**

Design and simulate fixed-point systems using Fixed-Point Designer™.

**See Also**

Inport | Out Bus Element
Topics
“Simplify Bus Interfaces”
“Ensure Outport is Virtual”
“Specify Data Types Using Data Type Assistant”

Introduced before R2006a
Parameter Writer

Write to a model instance parameter
Library: Simulink / Signal Routing

Description

The Parameter Writer block changes block parameter values in a referenced model by writing to instance parameters belonging to a Model block referencing a model.

Use the Parameter Writer block with the Initialize Function and Reset Function blocks to respond to events. For example, an event could be reading the value from a hardware sensor, and then updating a model parameter based on the sensor value.

Ports

Input

In — Parameter value
scalar | vector

Parameter value written to a block.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

Parameters

Parameter owner block — Show the parameter owner block
none (default)

Show the parameter owner block whose parameter value this block is writing. To change the parameter owner block, select a block from the Parameter Owner Selector Tree.
Programmatic Use
Block Parameter: ParameterOwnerBlock
Type: character vector
Value: '' | '<model path/block name>'
Default: ''

Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>bus</th>
<th>double</th>
<th>enumerated</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

See Also
Event Listener | Initialize Function | Reset Function | State Reader | State Writer | Terminate Function
Topics
“Using Initialize, Reset, and Terminate Functions”
“Create Test Harness to Generate Function Calls”
Initialize and Reset Parameter Values
“Initialize, Reset, and Terminate Function Limitations”
Permute Dimensions

Rearrange dimensions of multidimensional array dimensions

Library: Simulink / Math Operations

Description

The Permute Dimensions block reorders the elements of the input signal by permuting its dimensions. You specify the permutation to be applied to the input signal using the Order parameter.

For example, to transpose a 3-by-5 input signal, specify the permutation vector [2 1] for the Order parameter. When you do, the block reorders the elements of the input signal and outputs a 3-by-5 matrix.

You can use an array of buses as an input signal to a Permute Dimensions block. For details about defining and using an array of buses, see “Combine Buses into an Array of Buses”.

Ports

Input

Port_1 — Input signal

scalar | vector | matrix | N-D array

This port accepts scalar, vector, matrix, and N-dimensional signals of any data type that Simulink supports, including fixed-point, enumerated, and nonvirtual bus data types.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 |
        Boolean | fixed point | enumerated | bus
Output

Port_2 — Permutation of input signal
scalar | vector | matrix | N-D array

The block outputs the permutation of the input signal, according to the value of the Order parameter. The output has the same data type as the input.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus

Parameters

Order — Permutation vector
[2, 1] (default) | N-element vector, where N is the number of dimensions of the input signal

Specify the permutation order to apply to the dimensions of the input signal. The value of this parameter must be an N-element vector where N is the number of dimensions of the input signal. The elements of the permutation vector must be a rearrangement of the values from 1 to N.

For example, the permutation vector [2 1] applied to a 5-by-3 input signal results in a 3-by-5 output signal, in other words, the transpose of the input signal.

Programmatic Use
Block Parameter: Order
Type: character vector
Value: N-element vector
Default: ' [2 1] '

Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>bus</th>
<th>double</th>
<th>enumerated</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Permute Dimensions

<table>
<thead>
<tr>
<th>Multidimensional Signals</th>
<th>no</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variable-Size Signals</td>
<td>yes</td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
</tr>
</tbody>
</table>

Extended Capabilities

**C/C++ Code Generation**
Generate C and C++ code using Simulink® Coder™.

**PLC Code Generation**
Generate Structured Text code using Simulink® PLC Coder™.

**Fixed-Point Conversion**
Design and simulate fixed-point systems using Fixed-Point Designer™.

**See Also**
Math Function | permute

**Topics**
“Multidimensional Arrays” (MATLAB)

**Introduced in R2007a**
**PID Controller**

Continuous-time or discrete-time PID controller

**Library:** Simulink / Continuous

**Description**

The PID Controller block implements a PID controller (PID, PI, PD, P only, or I only). The block is identical to the Discrete PID Controller block with the **Time domain** parameter set to **Continuous-time**.

The block output is a weighted sum of the input signal, the integral of the input signal, and the derivative of the input signal. The weights are the proportional, integral, and derivative gain parameters. A first-order pole filters the derivative action.

The block supports several controller types and structures. Configurable options in the block include:

- Controller type (PID, PI, PD, P only, or I only) — See the **Controller** parameter.
- Controller form (Parallel or Ideal) — See the **Form** parameter.
- Time domain (continuous or discrete) — See the **Time domain** parameter.
- Initial conditions and reset trigger — See the **Source** and **External reset** parameters.
- Output saturation limits and built-in anti-windup mechanism — See the **Limit output** parameter.
- Signal tracking for bumpless control transfer and multiloop control — See the **Enable tracking mode** parameter.

As you change these options, the internal structure of the block changes by activating different variant subsystems. (See “Variant Subsystems” on page 15-495.) To examine the internal structure of the block and its variant subsystems, right-click the block and select **Mask > Look Under Mask**.
Control Configuration

In one common implementation, the PID Controller block operates in the feedforward path of a feedback loop.

![PID Controller block diagram](image)

The input of the block is typically an error signal, which is the difference between a reference signal and the system output. For a two-input block that permits setpoint weighting, see PID Controller (2DOF).

PID Gain Tuning

The PID controller coefficients are tunable either manually or automatically. Automatic tuning requires Simulink Control Design software. For more information about automatic tuning, see the **Select tuning method** parameter.

Ports

Input

**Port_1( u ) — Error signal input**

*scalar | vector*

Difference between a reference signal and the output of the system under control, as shown.
When the error signal is a vector, the block acts separately on each signal, vectorizing the PID coefficients and producing a vector output signal of the same dimensions. You can specify the PID coefficients and some other parameters as vectors of the same dimensions as the input signal. Doing so is equivalent to specifying a separate PID controller for each entry in the input signal.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

**P — Proportional gain**
scalar | vector

Proportional gain, provided from a source external to the block. External gain input is useful, for example, when you want to map a different PID parameterization to the PID gains of the block. You can also use external gain input to implement gain-scheduled PID control. In gain-scheduled control, you determine the PID coefficients by logic or other calculation in your model and feed them to the block.

**Dependencies**

To enable this port, set Controller parameters Source to external.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

**I — Integral gain**
scalar | vector

Integral gain, provided from a source external to the block. External gain input is useful, for example, when you want to map a different PID parameterization to the PID gains of the block. You can also use external gain input to implement gain-scheduled PID control.
In gain-scheduled control, you determine the PID coefficients by logic or other calculation in your model and feed them to the block.

When you supply gains externally, time variations in the integral gain are also integrated. This result occurs because of the way the PID gains are implemented within the block. For details, see the Controller parameters Source parameter.

Dependencies

To enable this port, set Controller parameters Source to external, and set Controller to a controller type that has integral action.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

D — Derivative gain

scalar | vector

Derivative gain, provided from a source external to the block. External gain input is useful, for example, when you want to map a different PID parameterization to the PID gains of the block. You can also use external gain input to implement gain-scheduled PID control. In gain-scheduled control, you determine the PID coefficients by logic or other calculation in your model and feed them to the block.

When you supply gains externally, time variations in the derivative gain are also differentiated. This result occurs because of the way the PID gains are implemented within the block. For details, see the Controller parameters Source parameter.

Dependencies

To enable this port, set Controller parameters Source to external, and set Controller to a controller type that has derivative action.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

N — Filter coefficient

scalar | vector

Derivative filter coefficient, provided from a source external to the block. External coefficient input is useful, for example, when you want to map a different PID parameterization to the PID gains of the block. You can also use the external input to implement gain-scheduled PID control. In gain-scheduled control, you determine the PID coefficients by logic or other calculation in your model and feed them to the block.
Dependencies

To enable this port, set **Controller parameters Source** to external, and set **Controller** to a controller type that has a filtered derivative.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

**Reset — External reset trigger**

scalar

Trigger to reset the integrator and filter to their initial conditions. The value of the **External reset** parameter determines whether reset occurs on a rising signal, a falling signal, or a level signal. The port icon indicates the selected trigger type. For example, the following illustration shows a continuous-time PID block with **External reset** set to rising.

![Diagram of PID block with External reset set to rising](image)

When the trigger occurs, the block resets the integrator and filter to the initial conditions specified by the **Integrator Initial condition** and **Filter Initial condition** parameters or the \(I_0\) and \(D_0\) ports.

**Note** To be compliant with the Motor Industry Software Reliability Association (MISRA) software standard, your model must use Boolean signals to drive the external reset ports of the PID controller block.

Dependencies

To enable this port, set **External reset** to any value other than none.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point | Boolean

**\(I_0\) — Integrator initial condition**

scalar | vector

Integrator initial condition, provided from a source external to the block.
Dependencies

To enable this port, set **Initial conditions Source** to **external**, and set **Controller** to a controller type that has integral action.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

**D₀** — **Filter initial condition**  
scalar | vector

Initial condition of the derivative filter, provided from a source external to the block.

Dependencies

To enable this port, set **Initial conditions Source** to **external**, and set **Controller** to a controller type that has derivative action.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

**TR** — **Tracking signal**  
scalar | vector

Signal for controller output to track. When signal tracking is active, the difference between the tracking signal and the block output is fed back to the integrator input.

Signal tracking is useful for implementing bumpless control transfer in systems that switch between two controllers. It can also be useful to prevent block windup in multiloop control systems. For more information, see the **Enable tracking mode** parameter.

Dependencies

To enable this port, select the **Enable tracking mode** parameter.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

Output

**Port_1( y )** — **Controller output**  
scalar | vector

Controller output, generally based on a sum of the input signal, the integral of the input signal, and the derivative of the input signal, weighted by the proportional, integral, and
derivative gain parameters. A first-order pole filters the derivative action. Which terms are present in the controller signal depends on what you select for the **Controller** parameter. The base controller transfer function for the current settings is displayed in the **Compensator formula** section of the block parameters and under the mask. Other parameters modify the block output, such as saturation limits specified by the **Upper Limit** and **Lower Limit** saturation parameters.

The controller output is a vector signal when any of the inputs is a vector signal. In that case, the block acts as $N$ independent PID controllers, where $N$ is the number of signals in the input vector.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | fixed point

## Parameters

**Controller — Controller type**

PID (default) | PI | PD | P | I

Specify which of the proportional, integral, and derivative terms are in the controller.

**PID**

- Proportional, integral, and derivative action.

**PI**

- Proportional and integral action only.

**PD**

- Proportional and derivative action only.

**P**

- Proportional action only.

**I**

- Integral action only.

**Tip** The controller transfer function for the current setting is displayed in the **Compensator formula** section of the block parameters and under the mask.
Programmatic Use

**Block Parameter:** Controller

**Type:** string, character vector

**Values:** "PID", "PI", "PD", "P", "I"

**Default:** "PID"

Form — Controller structure

Parallel (default) | Ideal

Specify whether the controller structure is parallel or ideal.

**Parallel**

The controller output is the sum of the proportional, integral, and derivative actions, weighted independently by $P$, $I$, and $D$, respectively. For example, for a continuous-time parallel-form PID controller, the transfer function is:

$$C_{\text{par}}(s) = P + I\left(\frac{1}{s}\right) + D\left(\frac{Ns}{s + N}\right).$$

For a discrete-time parallel-form controller, the transfer function is:

$$C_{\text{par}}(z) = P + I\alpha(z) + D\left[\frac{N}{1 + N\beta(z)}\right],$$

where the **Integrator method** and **Filter method** parameters determine $\alpha(z)$ and $\beta(z)$, respectively.

**Ideal**

The proportional gain $P$ acts on the sum of all actions. For example, for a continuous-time ideal-form PID controller, the transfer function is:

$$C_{\text{id}}(s) = P\left[1 + I\left(\frac{1}{s}\right) + D\left(\frac{Ns}{s + N}\right)\right].$$

For a discrete-time ideal-form controller, the transfer function is:

$$C_{\text{id}}(z) = P\left[1 + I\alpha(z) + D\frac{N}{1 + N\beta(z)}\right],$$

where the **Integrator method** and **Filter method** parameters determine $\alpha(z)$ and $\beta(z)$, respectively.
Tip The controller transfer function for the current settings is displayed in the Compensator formula section of the block parameters and under the mask.

Programmatic Use

Block Parameter: Controller
Type: string, character vector
Values: "Parallel", "Ideal"
Default: "Parallel"

Time domain — Specify continuous-time or discrete-time controller

Continuous-time (default) | Discrete-time

When you select Discrete-time, it is recommended that you specify an explicit sample time for the block. See the Sample time (-1 for inherited) parameter. Selecting Discrete-time also enables the Integrator method, and Filter method parameters.

When the PID Controller block is in a model with synchronous state control (see the State Control block), you cannot select Continuous-time.

Note The PID Controller and Discrete PID Controller blocks are identical except for the default value of this parameter.

Programmatic Use

Block Parameter: TimeDomain
Type: string, character vector
Values: "Continuous-time", "Discrete-time"
Default: "Continuous-time"

Sample time (-1 for inherited) — Discrete interval between samples

-1 (default) | positive scalar

Specify a sample time by entering a positive scalar value, such as 0.1. The default discrete sample time of -1 means that the block inherits its sample time from upstream blocks. However, it is recommended that you set the controller sample time explicitly, especially if you expect the sample time of upstream blocks to change. The effect of the controller coefficients P, I, D, and N depend on the sample time. Thus, for a given set of coefficient values, changing the sample time changes the performance of the controller.

See “Specify Sample Time” for more information.
To implement a continuous-time controller, set **Time domain** to Continuous-time.

**Tip** If you want to run the block with an externally specified or variable sample time, set this parameter to -1 and put the block in a Triggered Subsystem. Then, trigger the subsystem at the desired sample time.

**Dependencies**

To enable this parameter, set **Time domain** to Discrete-time.

**Programmatic Use**

**Block Parameter:** SampleTime  
**Type:** scalar  
**Values:** -1, positive scalar  
**Default:** -1

**Integrator method — Method for computing integral in discrete-time controller**  
**Forward Euler (default) | Backward Euler | Trapezoidal**

In discrete time, the integral term of the controller transfer function is $Ia(z)$, where $a(z)$ depends on the integrator method you specify with this parameter.

**Forward Euler**

Forward rectangular (left-hand) approximation,

$$a(z) = \frac{T_s}{z - 1}.$$  

This method is best for small sampling times, where the Nyquist limit is large compared to the bandwidth of the controller. For larger sampling times, the **Forward Euler** method can result in instability, even when discretizing a system that is stable in continuous time.

**Backward Euler**

Backward rectangular (right-hand) approximation,

$$a(z) = \frac{T_sz}{z - 1}.$$  

An advantage of the **Backward Euler** method is that discretizing a stable continuous-time system using this method always yields a stable discrete-time result.
Trapezoidal

Bilinear approximation,

\[ \alpha(z) = \frac{T_s z + 1}{2 z - 1}. \]

An advantage of the Trapezoidal method is that discretizing a stable continuous-time system using this method always yields a stable discrete-time result. Of all available integration methods, the Trapezoidal method yields the closest match between frequency-domain properties of the discretized system and the corresponding continuous-time system.

**Tip** The controller formula for the current setting is displayed in the **Compensator formula** section of the block parameters and under the mask.

**Note** For the Backward Euler or Trapezoidal methods, you cannot generate HDL code for the block if either:

- **Limit output** is selected and **Anti-Windup Method** is anything other than none.
- **Enable tracking mode** is selected.

For more information about discrete-time integration, see the Discrete-Time Integrator block reference page.

**Dependencies**

To enable this parameter, set **Time Domain** to Discrete-time and set **Controller** to a controller type with integral action.

**Programmatic Use**

**Block Parameter:** IntegratorMethod  
**Type:** string, character vector  
**Values:** "Forward Euler", "Backward Euler", "Trapezoidal"  
**Default:** "Forward Euler"

**Filter method — Method for computing derivative in discrete-time controller**

Forward Euler (default) | Backward Euler | Trapezoidal

In discrete time, the derivative term of the controller transfer function is:
\[ D \left[ \frac{N}{1 + Na(z)} \right] \]

where \( a(z) \) depends on the filter method you specify with this parameter.

**Forward Euler**
Forward rectangular (left-hand) approximation,

\[ a(z) = \frac{T_s}{z - 1}. \]

This method is best for small sampling times, where the Nyquist limit is large compared to the bandwidth of the controller. For larger sampling times, the Forward Euler method can result in instability, even when discretizing a system that is stable in continuous time.

**Backward Euler**
Backward rectangular (right-hand) approximation,

\[ a(z) = \frac{T_s z}{z - 1}. \]

An advantage of the Backward Euler method is that discretizing a stable continuous-time system using this method always yields a stable discrete-time result.

**Trapezoidal**
Bilinear approximation,

\[ a(z) = \frac{T_s z + 1}{2 \left( z - 1 \right)}. \]

An advantage of the Trapezoidal method is that discretizing a stable continuous-time system using this method always yields a stable discrete-time result. Of all available integration methods, the Trapezoidal method yields the closest match between frequency-domain properties of the discretized system and the corresponding continuous-time system.

**Tip** The controller formula for the current setting is displayed in the **Compensator formula** section of the block parameters and under the mask.
For more information about discrete-time integration, see the Discrete-Time Integrator block reference page.

**Dependencies**

To enable this parameter, set **Time Domain** to **Discrete-time** and set **Controller** to a controller type with derivative action.

**Programmatic Use**

**Block Parameter:** FilterMethod  
**Type:** string, character vector  
**Values:** "Forward Euler", "Backward Euler", "Trapezoidal"  
**Default:** "Forward Euler"

**Main**

**Source — Source for controller gains and filter coefficient**

**internal (default) | external**

Enabling external inputs for the parameters allows you to compute PID gains and filter coefficients externally to the block and provide them to the block as signal inputs.

**internal**

Specify the controller gains and filter coefficient using the block parameters **P, I, D**, and **N**.

**external**

Specify the PID gains and filter coefficient externally using block inputs. An additional input port appears on the block for each parameter that is required for the current controller type.

External gain input is useful, for example, when you want to map a different PID parameterization to the PID gains of the block. You can also use external gain input to implement gain-scheduled PID control. In gain-scheduled control, you determine the PID gains by logic or other calculation in your model and feed them to the block.

When you supply gains externally, time variations in the integral and derivative gain values are integrated and differentiated, respectively. This result occurs because in both continuous time and discrete time, the gains are applied to the signal before integration or differentiation. For example, for a continuous-time PID controller with external inputs, the integrator term is implemented as shown in the following illustration.
Within the block, the input signal \( u \) is multiplied by the externally supplied integrator gain, \( I \), before integration. This implementation yields:

\[
y_i = \int u I \, dt.
\]

Thus, the integrator gain is included in the integral. Similarly, in the derivative term of the block, multiplication by the derivative gain precedes the differentiation, which causes the derivative gain \( D \) to be differentiated.

**Programmatic Use**

**Block Parameter:** ControllerParametersSource  
**Type:** string, character vector  
**Values:** "internal", "external"  
**Default:** "internal"

**Proportional (P) — Proportional gain**  
1 (default) | scalar | vector

Specify a finite, real gain value for the proportional gain. When **Controller form** is:

- **Parallel** — Proportional action is independent of the integral and derivative actions. For instance, for a continuous-time parallel PID controller, the transfer function is:

\[
C_{\text{par}}(s) = P + I\left(\frac{1}{s}\right) + D\left(\frac{Ns}{s+N}\right).
\]

For a discrete-time parallel-form controller, the transfer function is:

\[
C_{\text{par}}(z) = P + I\alpha(z) + D\left[\frac{N}{1+N\beta(z)}\right],
\]

where the **Integrator method** and **Filter method** parameters determine \( \alpha(z) \) and \( \beta(z) \), respectively.

- **Ideal** — The proportional gain multiples the integral and derivative terms. For instance, for a continuous-time ideal PID controller, the transfer function is:
\[ C_{id}(s) = P \left[ 1 + I\left(\frac{1}{s}\right) + D\left(\frac{Ns}{s + N}\right) \right]. \]

For a discrete-time ideal-form controller, the transfer function is:

\[ C_{id}(z) = P \left[ 1 + I\alpha(z) + D\frac{N}{1 + N\beta(z)} \right], \]

where the Integrator method and Filter method parameters determine \( \alpha(z) \) and \( \beta(z) \), respectively.

**Tunable:** Yes

**Dependencies**

To enable this parameter, in the Main tab, set the controller-parameters Source to internal and set Controller to PID, PD, PI, or P.

**Programmatic Use**

**Block Parameter:** \( P \)

**Type:** scalar, vector

**Default:** 1

**Integral (I) — Integral gain**

1 (default) | scalar | vector

Specify a finite, real gain value for the integral gain.

**Tunable:** Yes

**Dependencies**

To enable this parameter, in the Main tab, set the controller-parameters Source to internal, and set Controller to a type that has integral action.

**Programmatic Use**

**Block Parameter:** \( I \)

**Type:** scalar, vector

**Default:** 1

**Derivative (D) — Derivative gain**

0 (default) | scalar | vector

Specify a finite, real gain value for the derivative gain.
**Tunable:** Yes

**Dependencies**

To enable this parameter, in the **Main** tab, set the controller-parameters **Source** to **internal**, and set **Controller** to **PID** or **PD**.

**Programmatic Use**
**Block Parameter:** D
**Type:** scalar, vector  
**Default:** 0

**Use filtered derivative — Apply filter to derivative term**
**on (default) | off**

For discrete-time PID controllers only, clear this option to replace the filtered derivative with an unfiltered discrete-time differentiator. When you do so, the derivative term of the controller transfer function becomes:

\[
\frac{Dz - 1}{zT_s}.
\]

For continuous-time PID controllers, the derivative term is always filtered.

**Dependencies**

To enable this parameter, set **Time domain** to **Discrete-time**, and set **Controller** to a type that has derivative action.

**Programmatic Use**
**Block Parameter:** UseFilter
**Type:** string, character vector
**Values:** "on", "off"
**Default:** "on"

**Filter coefficient (N) — Derivative filter coefficient**
100 (default) | scalar | vector

Specify a finite, real gain value for the filter coefficient. The filter coefficient determines the pole location of the filter in the derivative action of the block. The location of the filter pole depends on the **Time domain** parameter.

- When **Time domain** is **Continuous-time**, the pole location is \( s = -N \).
• When Time domain is Discrete-time, the pole location depends on the Filter method parameter.

<table>
<thead>
<tr>
<th>Filter Method</th>
<th>Location of Filter Pole</th>
</tr>
</thead>
<tbody>
<tr>
<td>Forward Euler</td>
<td>$z_{pole} = 1 - NT_s$</td>
</tr>
<tr>
<td>Backward Euler</td>
<td>$z_{pole} = \frac{1}{1 + NT_s}$</td>
</tr>
<tr>
<td>Trapezoidal</td>
<td>$z_{pole} = \frac{1 - NT_s/2}{1 + NT_s/2}$</td>
</tr>
</tbody>
</table>

The block does not support $N = \text{Inf}$ (ideal unfiltered derivative). When the Time domain is Discrete-time, you can clear Use filtered derivative to remove the derivative filter.

**Tunable:** Yes

**Dependencies**

To enable this parameter, in the Main tab, set the controller-parameters Source to internal and set Controller to PID or PD.

**Programmatic Use**

Block Parameter: $N$

Type: scalar, vector

Default: 100

**Select tuning method — Tool for automatic tuning of controller coefficients**

Transfer Function Based (PID Tuner App) (default) | Frequency Response Based

If you have Simulink Control Design software, you can automatically tune the PID coefficients. To do so, use this parameter to select a tuning tool, and click Tune.

Transfer Function Based (PID Tuner App)

Use PID Tuner, which lets you interactively tune PID coefficients while examining relevant system responses to validate performance. By default, PID Tuner works with a linearization of your plant model. For models that cannot be linearized, you can tune PID coefficients against a plant model estimated from simulated or measured response data. For more information, see “Introduction to Model-Based PID Tuning in Simulink” (Simulink Control Design).
Frequency Response Based

Use **Frequency Response Based PID Tuner**, which tunes PID controller coefficients based on frequency-response estimation data obtained by simulation. This tuning approach is especially useful for plants that are not linearizable or that linearize to zero. For more information, see “Design PID Controller from Plant Frequency-Response Data” (Simulink Control Design).

Both of these tuning methods assume a single-loop control configuration. Simulink Control Design software includes other tuning approaches that suit more complex configurations. For information about other ways to tune a PID Controller block, see “Choose a Control Design Approach” (Simulink Control Design).

**Enable zero-crossing detection — Detect zero crossings on reset and on entering or leaving a saturation state**

*on* (default) | *off*

Zero-crossing detection can accurately locate signal discontinuities without resorting to excessively small time steps that can lead to lengthy simulation times. If you select **Limit output** or activate **External reset** in your PID Controller block, activating zero-crossing detection can reduce computation time in your simulation. Selecting this parameter activates zero-crossing detection:

- At initial-state reset
- When entering an upper or lower saturation state
- When leaving an upper or lower saturation state

For more information about zero-crossing detection, see “Zero-Crossing Detection”.

**Programmatic Use**

**Block Parameter:** ZeroCross

**Type:** string, character vector

**Values:** "on", "off"

**Default:** "on"

**Initialization**

**Source — Source for integrator and derivative initial conditions**

*internal* (default) | *external*

Simulink uses initial conditions to initialize the integrator and derivative-filter (or the unfiltered derivative) output at the start of a simulation or at a specified trigger event.
(See the External reset parameter.) These initial conditions determine the initial block output. Use this parameter to select how to supply the initial condition values to the block.

**internal**

Specify the initial conditions using the Integrator Initial condition and Filter Initial condition parameters. If Use filtered derivative is not selected, use the Differentiator parameter to specify the initial condition for the unfiltered differentiator instead of a filter initial condition.

**external**

Specify the initial conditions externally using block inputs. Additional input ports I₀ and D₀ appear on the block. If Use filtered derivative is not selected, supply the initial condition for the unfiltered differentiator at D₀ instead of a filter initial condition.

**Programmatic Use**

**Block Parameter:** InitialConditionSource  
**Type:** string, character vector  
**Values:** "internal", "external"  
**Default:** "internal"

**Integrator — Integrator initial condition**

0 (default) | scalar | vector

Simulink uses the integrator initial condition to initialize the integrator at the start of a simulation or at a specified trigger event (see External reset). The integrator initial condition and the filter initial condition determine the initial output of the PID controller block.

The integrator initial condition cannot be NaN or Inf.

**Dependencies**

To use this parameter, in the Initialization tab, set Source to internal, and set Controller to a type that has integral action.

**Programmatic Use**

**Block Parameter:** InitialConditionForIntegrator  
**Type:** scalar, vector  
**Default:** 0
Filter — Filter initial condition
0 (default) | scalar | vector

Simulink uses the filter initial condition to initialize the derivative filter at the start of a simulation or at a specified trigger event (see External reset). The integrator initial condition and the filter initial condition determine the initial output of the PID controller block.

The filter initial condition cannot be NaN or Inf.

Dependencies
To use this parameter, in the Initialization tab, set Source to internal, and use a controller that has a derivative filter.

Programmatic Use
Block Parameter: InitialConditionForFilter
Type: scalar, vector
Default: 0

Differentiator — Initial condition for unfiltered derivative
0 (default) | scalar | vector

When you use an unfiltered derivative, Simulink uses this parameter to initialize the differentiator at the start of a simulation or at a specified trigger event (see External reset). The integrator initial condition and the derivative initial condition determine the initial output of the PID controller block.

The derivative initial condition cannot be NaN or Inf.

Dependencies
To use this parameter, set Time domain to Discrete-time, clear the Use filtered derivative check box, and in the Initialization tab, set Source to internal.

Programmatic Use
Block Parameter: DifferentiatorICPrevScaledInput
Type: scalar, vector
Default: 0

Initial condition setting — Location at which initial condition is applied
State (most efficient) (default) | Output

Use this parameter to specify whether to apply the Integrator Initial condition and Filter Initial condition parameter to the corresponding block state or output. You can
change this parameter at the command line only, using `set_param` to set the InitialConditionSetting parameter of the block.

State (most efficient)

Use this option in all situations except when the block is in a triggered subsystem or a function-call subsystem and simplified initialization mode is enabled.

Output

Use this option when the block is in a triggered subsystem or a function-call subsystem and simplified initialization mode is enabled.

For more information about the **Initial condition setting** parameter, see the Discrete-Time Integrator block.

This parameter is only accessible through programmatic use.

**Programmatic Use**

**Block Parameter:** InitialConditionSetting  
**Type:** string, character vector  
**Values:** "state", "output"  
**Default:** "state"

**External reset — Trigger for resetting integrator and filter values**

none (default) | rising | falling | either | level

Specify the trigger condition that causes the block to reset the integrator and filter to initial conditions. (If **Use filtered derivative** is not selected, the trigger resets the integrator and differentiator to initial conditions.) Selecting any option other than none enables the Reset port on the block for the external reset signal.

- **none**
  The integrator and filter (or differentiator) outputs are set to initial conditions at the beginning of simulation, and are not reset during simulation.

- **rising**
  Reset the outputs when the reset signal has a rising edge.

- **falling**
  Reset the outputs when the reset signal has a falling edge.

- **either**
  Reset the outputs when the reset signal either rises or falls.
level

Reset the outputs when the reset signal either:

- Is nonzero at the current time step
- Changes from nonzero at the previous time step to zero at the current time step

This option holds the outputs to the initial conditions while the reset signal is nonzero.

**Dependencies**

To enable this parameter, set **Controller** to a type that has derivative or integral action.

**Programmatic Use**

**Block Parameter:** ExternalReset  
**Type:** string, character vector  
**Values:** "none", "rising", "falling", "either", "level"  
**Default:** "none"

**Ignore reset when linearizing — Force linearization to ignore reset**  
off (default) | on

Select to force Simulink and Simulink Control Design linearization commands to ignore any reset mechanism specified in the **External reset** parameter. Ignoring reset states allows you to linearize a model around an operating point even if that operating point causes the block to reset.

**Programmatic Use**

**Block Parameter:** IgnoreLimit  
**Type:** string, character vector  
**Values:** "off", "on"  
**Default:** "off"

**Enable tracking mode — Activate signal tracking**  
off (default) | on

Signal tracking lets the block output follow a tracking signal that you provide at the **TR** port. When signal tracking is active, the difference between the tracking signal and the block output is fed back to the integrator input with a gain **Kt**, specified by the **Tracking gain (Kt)** parameter. Signal tracking has several applications, including bumpless control transfer and avoiding windup in multiloop control structures.
**Bumpless control transfer**

Use signal tracking to achieve bumpless control transfer in systems that switch between two controllers. Suppose you want to transfer control between a PID controller and another controller. To do so, connecting the controller output to the TR input as shown in the following illustration.

![Diagram of Bumpless Control Transfer](image)

For more information, see “Bumpless Control Transfer” on page 14-128.

**Multiloop control**

Use signal tracking to prevent block windup in multiloop control approaches, as in the following model.

![Diagram of Multiloop Control](image)

The Inner Loop subsystem contains the blocks shown in the following diagram.
Because the PID controller tracks the output of the inner loop, its output never exceeds the saturated inner-loop output. For more details, see “Prevent Block Windup in Multiloop Control” on page 14-127.

**Dependencies**

To enable this parameter, set **Controller** to a type that has integral action.

**Programmatic Use**

**Block Parameter:** TrackingMode  
**Type:** string, character vector  
**Values:** "off", "on"  
**Default:** "off"

**Tracking coefficient (Kt) — Gain of signal-tracking feedback loop**  
1 (default) | scalar

When you select **Enable tracking mode**, the difference between the signal TR and the block output is fed back to the integrator input with a gain Kt. Use this parameter to specify the gain in that feedback loop.

**Dependencies**

To enable this parameter, select **Enable tracking mode**.

**Programmatic Use**

**Block Parameter:** Kt  
**Type:** scalar  
**Default:** 1

**Output saturation**

**Limit Output — Limit block output to specified saturation values**  
off (default) | on
Activating this option limits the block output internally to the block, so that you do not need a separate Saturation on page 1-1704 block after the controller. It also allows you to activate the anti-windup mechanism built into the block (see the Anti-windup method parameter). Specify the saturation limits using the **Lower saturation limit** and **Upper saturation limit** parameters.

**Programmatic Use**

**Block Parameter:** LimitOutput  
**Type:** string, character vector  
**Values:** "off", "on"  
**Default:** "off"

**Upper limit — Upper saturation limit for block output**  
Inf (default) | scalar

Specify the upper limit for the block output. The block output is held at the **Upper saturation limit** whenever the weighted sum of the proportional, integral, and derivative actions exceeds that value.

**Dependencies**

To enable this parameter, select **Limit output**.

**Programmatic Use**

**Block Parameter:** UpperSaturationLimit  
**Type:** scalar  
**Default:** Inf

**Lower limit — Lower saturation limit for block output**  
-Inf (default) | scalar

Specify the lower limit for the block output. The block output is held at the **Lower saturation limit** whenever the weighted sum of the proportional, integral, and derivative actions goes below that value.

**Dependencies**

To enable this parameter, select **Limit output**.

**Programmatic Use**

**Block Parameter:** LowerSaturationLimit  
**Type:** scalar  
**Default:** -Inf
Ignore saturation when linearizing — Force linearization to ignore output limits
off (default) | on

Force Simulink and Simulink Control Design linearization commands to ignore block output limits specified in the **Upper limit** and **Lower limit** parameters. Ignoring output limits allows you to linearize a model around an operating point even if that operating point causes the block to exceed the output limits.

**Dependencies**

To enable this parameter, select the **Limit output** parameter.

**Programmatic Use**
Block Parameter: LinearizeAsGain
Type: string, character vector
Values: "off", "on"
Default: "off"

Anti-windup method — Integrator anti-windup method
none (default) | back-calculation | clamping

When you select **Limit output** and the weighted sum of the controller components exceeds the specified output limits, the block output holds at the specified limit. However, the integrator output can continue to grow (integrator windup), increasing the difference between the block output and the sum of the block components. In other words, the internal signals in the block can be unbounded even if the output appears bounded by saturation limits. Without a mechanism to prevent integrator windup, two results are possible:

- If the sign of the input signal never changes, the integrator continues to integrate until it overflows. The overflow value is the maximum or minimum value for the data type of the integrator output.
- If the sign of the input signal changes once the weighted sum has grown beyond the output limits, it can take a long time to unwind the integrator and return the weighted sum within the block saturation limit.

In either case, controller performance can suffer. To combat the effects of windup without an anti-windup mechanism, it may be necessary to detune the controller (for example, by reducing the controller gains), resulting in a sluggish controller. To avoid this problem, activate an anti-windup mechanism using this parameter.
none

Do not use an anti-windup mechanism.

back-calculation

Unwind the integrator when the block output saturates by feeding back to the integrator the difference between the saturated and unsaturated control signal. The following diagram represents the back-calculation feedback circuit for a continuous-time controller. To see the actual feedback circuit for your controller configuration, right-click on the block and select Mask > Look Under Mask.

Use the Back-calculation coefficient ($K_b$) parameter to specify the gain of the anti-windup feedback circuit. It is usually satisfactory to set $K_b = I$, or for controllers with derivative action, $K_b = \sqrt{I \cdot D}$. Back-calculation can be effective for plants with relatively large dead time [1].

clamping

Integration stops when the sum of the block components exceeds the output limits and the integrator output and block input have the same sign. Integration resumes when the sum of the block components exceeds the output limits and the integrator output and block input have opposite sign. Clamping is sometimes referred to as conditional integration.

Clamping can be useful for plants with relatively small dead times, but can yield a poor transient response for large dead times [1].
Dependencies

To enable this parameter, select the Limit output parameter.

Programmatic Use
Block Parameter: AntiWindupMode
Type: string, character vector
Values: "none", "back-calculation", "clamping"
Default: "none"

Back-calculation coefficient (Kb) — Gain coefficient of anti-windup feedback loop
1 (default) | scalar

The back-calculation anti-windup method unwinds the integrator when the block output saturates. It does so by feeding back to the integrator the difference between the saturated and unsaturated control signal. Use the Back-calculation coefficient (Kb) parameter to specify the gain of the anti-windup feedback circuit. For more information, see the Anti-windup method parameter.

Dependencies

To enable this parameter, select the Limit output parameter, and set the Anti-windup method parameter to back-calculation.

Programmatic Use
Block Parameter: Kb
Type: scalar
Default: 1

Data Types

The parameters in this tab are primarily of use in fixed-point code generation using Fixed-Point Designer. They define how numeric quantities associated with the block are stored and processed when you generate code.

If you need to configure data types for fixed-point code generation, click Open Fixed-Point Tool and use that tool to configure the rest of the parameters in the tab. For information about using Fixed-Point Tool, see “Autoscaling Data Objects Using the Fixed-Point Tool” (Fixed-Point Designer).
After you use Fixed-Point Tool, you can use the parameters in this tab to make adjustments to fixed-point data-type settings if necessary. For each quantity associated with the block, you can specify:

- Floating-point or fixed-point data type, including whether the data type is inherited from upstream values in the block.
- The minimum and maximum values for the quantity, which determine how the quantity is scaled for fixed-point representation.

For assistance in selecting appropriate values, click to open the Data Type Assistant for the corresponding quantity. For more information, see “Specify Data Types Using Data Type Assistant”.

![Fixed-Point Tool parameters](image)
The specific quantities listed in the Data Types tab vary depending on how you configure the PID controller block. In general, you can configure data types for the following types of quantities:

- **Product output** — Stores the result of a multiplication carried out under the block mask. For example, **P product output** stores the output of the gain block that multiplies the block input with the proportional gain \( P \).

- **Parameter** — Stores the value of a numeric block parameter, such as \( P \), \( I \), or \( D \).

- **Block output** — Stores the output of a block that resides under the PID controller block mask. For example, use **Integrator output** to specify the data type of the output of the block called Integrator. This block resides under the mask in the Integrator subsystem, and computes integrator term of the controller action.

- **Accumulator** — Stores values associated with a sum block. For example, **SumI2 Accumulator** sets the data type of the accumulator associated with the sum block SumI2. This block resides under the mask in the Back Calculation subsystem of the Anti-Windup subsystem.

In general, you can find the block associated with any listed parameter by looking under the PID Controller block mask and examining its subsystems. You can also use the Model Explorer to search under the mask for the listed parameter name, such as **SumI2**. (See “Search and Edit Using Model Explorer”.)

### Matching Input and Internal Data Types

By default, all data types in the block are set to **Inherit: Inherit via internal rule**. With this setting, Simulink chooses data types to balance numerical accuracy, performance, and generated code size, while accounting for the properties of the embedded target hardware.

Under some conditions, incompatibility can occur between data types within the block. For instance, in continuous time, the Integrator block under the mask can accept only signals of type **double**. If the block input signal is a type that cannot be converted to **double**, such as uint16, the internal rules for type inheritance generate an error when you generate code.

To avoid such errors, you can use the Data Types settings to force a data type conversion. For instance, you can explicitly set **P product output**, **I product output**, and **D product output** to **double**, ensuring that the signals reaching the continuous-time integrators are of type **double**.
In general, it is not recommended to use the block in continuous time for code generation applications. However, similar data type errors can occur in discrete time, if you explicitly set some values to data types that are incompatible with downstream signal constraints within the block. In such cases, use the Data Types settings to ensure that all data types are internally compatible.

**Fixed-Point Operational Parameters**

**Integer rounding mode — Rounding mode for fixed-point operations**

Floor (default) | Ceiling | Convergent | Nearest | Round | Simplest | Zero

Specify the rounding mode for fixed-point operations. For more information, see “Rounding” (Fixed-Point Designer).

Block parameters always round to the nearest representable value. To control the rounding of a block parameter, enter an expression using a MATLAB rounding function into the mask field.

**Programmatic Use**

**Block Parameter:** RndMeth

**Type:** character vector

**Values:** 'Ceiling' | 'Convergent' | 'Floor' | 'Nearest' | 'Round' | 'Simplest' | 'Zero'

**Default:** 'Floor'

**Saturate on integer overflow — Method of overflow action**

off (default) | on

Specify whether overflows saturate or wrap.

- **off** — Overflows wrap to the appropriate value that the data type can represent.
  
  For example, the number 130 does not fit in a signed 8-bit integer and wraps to -126.

- **on** — Overflows saturate to either the minimum or maximum value that the data type can represent.
  
  For example, an overflow associated with a signed 8-bit integer can saturate to -128 or 127.

**Tip**

1-1436
• Consider selecting this check box when your model has a possible overflow and you want explicit saturation protection in the generated code.
• Consider clearing this check box when you want to optimize efficiency of your generated code.

Clearing this check box also helps you to avoid overspecifying how a block handles out-of-range signals. For more information, see “Check for Signal Range Errors”.
• When you select this check box, saturation applies to every internal operation on the block, not just the output or result.
• In general, the code generation process can detect when overflow is not possible. In this case, the code generator does not produce saturation code.

**Programmatic Use**

**Block Parameter:** SaturateOnIntegerOverflow

*Type:* character vector  
*Values:* 'off' | 'on'  
*Default:* 'off'

**Lock data type settings against changes by the fixed-point tools — Prevent fixed-point tools from overriding data types**

*off* (default) | *on*

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on this block. For more information, see “Lock the Output Data Type Setting” (Fixed-Point Designer).

**Programmatic Use**

**Block Parameter:** LockScale

*Type:* character vector  
*Values:* 'off' | 'on'  
*Default:* 'off'

**State Attributes**

The parameters in this tab are primarily of use in code generation.

**State name (e.g., 'position') — Name for continuous-time filter and integrator states**

'*' (default) | character vector
Assign a unique name to the state associated with the integrator or the filter, for continuous-time PID controllers. (For information about state names in a discrete-time PID controller, see the State name parameter.) The state name is used, for example:

- For the corresponding variable in generated code
- As part of the storage name when logging states during simulation
- For the corresponding state in a linear model obtain by linearizing the block

A valid state name begins with an alphabetic or underscore character, followed by alphanumeric or underscore characters.

Dependencies

To enable this parameter, set Time domain to Continuous-time.

Programmatic Use

Parameter: IntegratorContinuousStateAttributes, FilterContinuousStateAttributes
Type: character vector
Default: ''

State name — Names for discrete-time filter and integrator states
empty string (default) | string | character vector

Assign a unique name to the state associated with the integrator or the filter, for discrete-time PID controllers. (For information about state names in a continuous-time PID controller, see the State name (e.g., 'position') parameter.)

A valid state name begins with an alphabetic or underscore character, followed by alphanumeric or underscore characters. The state name is used, for example:

- For the corresponding variable in generated code
- As part of the storage name when logging states during simulation
- For the corresponding state in a linear model obtain by linearizing the block

For more information about the use of state names in code generation, see “Apply Storage Classes to Individual Signal, State, and Parameter Data Elements” (Simulink Coder).

Dependencies

To enable this parameter, set Time domain to Discrete-time.
Programmatic Use

Parameter: IntegratorStateIdentifier, FilterStateIdentifier

Type: string, character vector

Default: ""

State name must resolve to Simulink signal object — Require that state name resolve to a signal object

off (default) | on

Select this parameter to require that the discrete-time integrator or filter state name resolves to a Simulink signal object.

Dependencies

To enable this parameter for the discrete-time integrator or filter state:

1. Set Time domain to Discrete-time.
2. Specify a value for the integrator or filter State name.
3. Set the model configuration parameter Signal resolution to a value other than None.

Selecting this check box disables Code generation storage class for the corresponding integrator or filter state.

Programmatic Use

Block Parameter: IntegratorStateMustResolveToSignalObject, FilterStateMustResolveToSignalObject

Type: string, character vector

Values: "off", "on"

Default: "off"

Code generation storage class — Storage class for code generation

Auto (default) | ExportedGlobal | ImportedExtern | ImportedExternPointer

Select state storage class for code generation. If you do not need to interface to external code, select Auto.

For more information, see “Apply Storage Classes to Individual Signal, State, and Parameter Data Elements” (Simulink Coder) and “Apply Built-In and Customized Storage Classes to Data Elements” (Embedded Coder).
Dependencies

To enable this parameter for the discrete-time integrator or filter state:

1. Set **Time domain** to **Discrete-time**.
2. Specify a value for the integrator or filter **State name**.
3. Set the model configuration parameter **Signal resolution** to a value other than None.

**Programmatic Use**

**Block Parameter:** IntegratorRTWStateStorageClass, FilterRTWStateStorageClass  
**Type:** string, character vector  
**Values:** "Auto", "ExportedGlobal", "ImportedExtern" | "ImportedExternPointer"  
**Default:** "Auto"

**Code generation storage type qualifier — Storage type qualifier**

empty string (default) | character vector | "const" | "volatile" | ...

Specify a storage type qualifier such as const or volatile.

**Note** This parameter will be removed in a future release. To apply storage type qualifiers to data, use custom storage classes and memory sections. Unless you use an ERT-based code generation target with Embedded Coder, custom storage classes and memory sections do not affect the generated code.

**Dependencies**

To enable this parameter, set **Code generation storage class** to any value other than Auto.

**Programmatic Use**

**Block Parameter:** IntegratorRTWStateStorageTypeQualifier, FilterRTWStateStorageTypeQualifier  
**Type:** string, character vector  
**Values:** "" | "const", "volatile"  
**Default:** ""
### Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>double</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### References


### Extended Capabilities

#### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

For continuous-time PID controllers *(Time domain set to Continuous-time)*:

- Consider using “Model Discretizer” to map continuous-time blocks to discrete equivalents that support code generation. To access Model Discretizer, from your model, in the Apps tab, under Control Systems, click Model Discretizer.
- Not recommended for production code.

For discrete-time PID controllers *(Time domain set to Discrete-time)*:

- Depends on absolute time when placed inside a triggered subsystem hierarchy.
- Generated code relies on memcpy or memset functions (string.h) under certain conditions.
**HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

**HDL Architecture**

This block has a single, default HDL architecture.

**HDL Block Properties**

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConstrainedOutputPipeline</td>
<td>Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. For more details, see “ConstrainedOutputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td>InputPipeline</td>
<td>Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “InputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td>OutputPipeline</td>
<td>Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “OutputPipeline” (HDL Coder).</td>
</tr>
</tbody>
</table>

**Restrictions**

- HDL code generation is supported for discrete-time PID controllers only (Time domain set to Discrete-time).
- If the Integrator method is set to BackwardEuler or Trapezoidal, you cannot generate HDL code for the block under either of the following conditions:
  - Limit output is selected and the Anti-Windup Method is anything other than none.
  - Enable tracking mode is selected.
- HDL code generation does not support these settings:
  - Continuous-time.
• **Source** > **external**.
• **External reset** > **rising, falling, either**, or **level**.
• **Anti-windup method** **clamping** when you use double inputs.

**PLC Code Generation**
Generate Structured Text code using Simulink® PLC Coder™.

**Fixed-Point Conversion**
Design and simulate fixed-point systems using Fixed-Point Designer™.

Fixed-point code generation is supported for discrete-time PID controllers only (**Time domain** set to **Discrete-time**).

**See Also**
Derivative | Discrete PID Controller | Gain | Integrator | PID Controller (2DOF)

*Introduced in R2009b*
**PID Controller (2DOF)**

Continuous-time or discrete-time two-degree-of-freedom PID controller

**Library:** Simulink / Continuous

---

**Description**

The PID Controller (2DOF) block implements a two-degree-of-freedom PID controller (PID, PI, or PD). The block is identical to the Discrete PID Controller (2DOF) block with the **Time domain** parameter set to Continuous-time.

The block generates an output signal based on the difference between a reference signal and a measured system output. The block computes a weighted difference signal for the proportional and derivative actions according to the setpoint weights (b and c) that you specify. The block output is the sum of the proportional, integral, and derivative actions on the respective difference signals, where each action is weighted according to the gain parameters P, I, and D. A first-order pole filters the derivative action.

The block supports several controller types and structures. Configurable options in the block include:

- Controller type (PID, PI, or PD) — See the **Controller** parameter.
- Controller form (Parallel or Ideal) — See the **Form** parameter.
- Time domain (continuous or discrete) — See the **Time domain** parameter.
- Initial conditions and reset trigger — See the **Source** and **External reset** parameters.
- Output saturation limits and built-in anti-windup mechanism — See the **Limit output** parameter.
- Signal tracking for bumpless control transfer and multiloop control — See the **Enable tracking mode** parameter.

As you change these options, the internal structure of the block changes by activating different variant subsystems. (See “Variant Subsystems” on page 15-495.) To examine
the internal structure of the block and its variant subsystems, right-click the block and select **Mask > Look Under Mask**.

**Control Configuration**

In one common implementation, the PID Controller block operates in the feedforward path of a feedback loop.

For a single-input block that accepts an error signal (a difference between a setpoint and a system output), see PID Controller.

**PID Gain Tuning**

The PID controller coefficients and the setpoint weights are tunable either manually or automatically. Automatic tuning requires Simulink Control Design software. For more information about automatic tuning, see the **Select tuning method** parameter.

**Ports**

**Input**

**Ref** — Reference signal

* scalar | vector

Reference signal for plant to follow, as shown.
When the reference signal is a vector, the block acts separately on each signal, vectorizing the PID coefficients and producing a vector output signal of the same dimensions. You can specify the PID coefficients and some other parameters as vectors of the same dimensions as the input signal. Doing so is equivalent to specifying a separate PID controller for each entry in the input signal.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

Port_1( y ) — Measured system output
scalar | vector

Feedback signal for the controller, from the plant output.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

P — Proportional gain
scalar | vector

Proportional gain, provided from a source external to the block. External gain input is useful, for example, when you want to map a different PID parameterization to the PID gains of the block. You can also use external gain input to implement gain-scheduled PID control. In gain-scheduled control, you determine the PID coefficients by logic or other calculation in your model and feed them to the block.

Dependencies

To enable this port, set Controller parameters Source to external.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point
**I — Integral gain**
scalar | vector

Integral gain, provided from a source external to the block. External gain input is useful, for example, when you want to map a different PID parameterization to the PID gains of the block. You can also use external gain input to implement gain-scheduled PID control. In gain-scheduled control, you determine the PID coefficients by logic or other calculation in your model and feed them to the block.

When you supply gains externally, time variations in the integral gain are also integrated. This result occurs because of the way the PID gains are implemented within the block. For details, see the Controller parameters Source parameter.

**Dependencies**
To enable this port, set Controller parameters Source to external, and set Controller to a controller type that has integral action.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

**D — Derivative gain**
scalar | vector

Derivative gain, provided from a source external to the block. External gain input is useful, for example, when you want to map a different PID parameterization to the PID gains of the block. You can also use external gain input to implement gain-scheduled PID control. In gain-scheduled control, you determine the PID coefficients by logic or other calculation in your model and feed them to the block.

When you supply gains externally, time variations in the derivative gain are also differentiated. This result occurs because of the way the PID gains are implemented within the block. For details, see the Controller parameters Source parameter.

**Dependencies**
To enable this port, set Controller parameters Source to external, and set Controller to a controller type that has derivative action.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

**N — Filter coefficient**
scalar | vector
Derivative filter coefficient, provided from a source external to the block. External coefficient input is useful, for example, when you want to map a different PID parameterization to the PID gains of the block. You can also use the external input to implement gain-scheduled PID control. In gain-scheduled control, you determine the PID coefficients by logic or other calculation in your model and feed them to the block.

**Dependencies**

To enable this port, set **Controller parameters Source** to external, and set **Controller** to a controller type that has a filtered derivative.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | fixed point

**b — Proportional setpoint weight**

scalar | vector

Proportional setpoint weight, provided from a source external to the block. External input is useful, for example, when you want to map a different PID parameterization to the PID gains of the block. You can also use the external input to implement gain-scheduled PID control. In gain-scheduled control, you determine the PID coefficients by logic or other calculation in your model and feed them to the block.

**Dependencies**

To enable this port, set **Controller parameters Source** to external.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | fixed point

**c — Derivative setpoint weight**

scalar | vector

Derivative setpoint weight, provided from a source external to the block. External input is useful, for example, when you want to map a different PID parameterization to the PID gains of the block. You can also use the external input to implement gain-scheduled PID control. In gain-scheduled control, you determine the PID coefficients by logic or other calculation in your model and feed them to the block.

**Dependencies**

To enable this port, set **Controller parameters Source** to external, and set **Controller** to a controller type that has derivative action.
Reset — External reset trigger

 scalar

Trigger to reset the integrator and filter to their initial conditions. Use the External reset parameter to specify what kind of signal triggers a reset. The port icon indicates the trigger type specified in that parameter. For example, the following illustration shows a continuous-time PID Controller (2DOF) block with External reset set to rising.

When the trigger occurs, the block resets the integrator and filter to the initial conditions specified by the Integrator Initial condition and Filter Initial condition parameters or the $I_0$ and $D_0$ ports.

**Note** To be compliant with the Motor Industry Software Reliability Association (MISRA) software standard, your model must use Boolean signals to drive the external reset ports of the PID controller block.

**Dependencies**

To enable this port, set External reset to any value other than none.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point | Boolean

$I_0$ — Integrator initial condition

 scalar | vector

Integrator initial condition, provided from a source external to the block.
Dependencies

To enable this port, set **Initial conditions Source** to external, and set **Controller** to a controller type that has integral action.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

\( D_0 \) — Filter initial condition
scalar | vector

Initial condition of the derivative filter, provided from a source external to the block.

Dependencies

To enable this port, set **Initial conditions Source** to external, and set **Controller** to a controller type that has derivative action.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

\( TR \) — Tracking signal
scalar | vector

Signal for controller output to track. When signal tracking is active, the difference between the tracking signal and the block output is fed back to the integrator input. Signal tracking is useful for implementing bumpless control transfer in systems that switch between two controllers. It can also be useful to prevent block windup in multiloop control systems. For more information, see the **Enable tracking mode** parameter.

Dependencies

To enable this port, select the **Enable tracking mode** parameter.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

**Output**

\( \text{Port}_1(u) \) — Controller output
scalar | vector

Controller output, generally based on a sum of the input signal, the integral of the input signal, and the derivative of the input signal, weighted by the setpoint weights and by the
proportional, integral, and derivative gain parameters. A first-order pole filters the derivative action. Which terms are present in the controller signal depends on what you select for the Controller parameter. The base controller transfer function for the current settings is displayed in the Compensator formula section of the block parameters and under the mask. Other parameters modify the block output, such as saturation limits specified by the Upper Limit and Lower Limit saturation parameters.

The controller output is a vector signal when any of the inputs is a vector signal. In that case, the block acts as \( N \) independent PID controllers, where \( N \) is the number of signals in the input vector.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

**Parameters**

**Controller — Controller type**

PID (default) | PI | PD

Specify which of the proportional, integral, and derivative terms are in the controller.

PID

- Proportional, integral, and derivative action.

PI

- Proportional and integral action only.

PD

- Proportional and derivative action only.

**Tip** The controller output for the current setting is displayed in the Compensator formula section of the block parameters and under the mask.

**Programmatic Use**

**Block Parameter:** Controller

**Type:** string, character vector

**Values:** "PID", "PI", "PD"

**Default:** "PID"
Form — Controller structure
Parallel (default) | Ideal

Specify whether the controller structure is parallel or ideal.

Parallel
The proportional, integral, and derivative gains $P$, $I$, and $D$, are applied independently. For example, for a continuous-time 2-DOF PID controller in parallel form, the controller output $u$ is:

$$u = P(br - y) + I \frac{1}{s}(r - y) + D \frac{N}{1 + N \frac{1}{s}}(cr - y),$$

where $r$ is the reference signal, $y$ is the measured plant output signal, and $b$ and $c$ are the setpoint weights.

For a discrete-time 2-DOF controller in parallel form, the controller output is:

$$u = P(br - y) + I\alpha(z)(r - y) + D \frac{N}{1 + N\beta(z)}(cr - y),$$

where the Integrator method and Filter method parameters determine $\alpha(z)$ and $\beta(z)$, respectively.

Ideal
The proportional gain $P$ acts on the sum of all actions. For example, for a continuous-time 2-DOF PID controller in ideal form, the controller output is:

$$u = P \left[ (br - y) + I \frac{1}{s}(r - y) + D \frac{N}{1 + N \frac{1}{s}}(cr - y) \right].$$

For a discrete-time 2-DOF PID controller in ideal form, the transfer function is:

$$u = P \left[ (br - y) + I\alpha(z)(r - y) + D \frac{N}{1 + N\beta(z)}(cr - y) \right],$$

where the Integrator method and Filter method parameters determine $\alpha(z)$ and $\beta(z)$, respectively.

Tip The controller output for the current settings is displayed in the Compensator formula section of the block parameters and under the mask.
Programmatic Use
Block Parameter: Controller
Type: string, character vector
Values: "Parallel", "Ideal"
Default: "Parallel"

**Time domain — Specify continuous-time or discrete-time controller**
Continuous-time (default) | Discrete-time

When you select Discrete-time, it is recommended that you specify an explicit sample time for the block. See the **Sample time (-1 for inherited)** parameter. Selecting Discrete-time also enables the **Integrator method**, and **Filter method** parameters.

When the PID Controller block is in a model with synchronous state control (see the State Control block), you cannot select Continuous-time.

**Note** The PID Controller (2DOF) and Discrete PID Controller (2DOF) blocks are identical except for the default value of this parameter.

Programmatic Use
Block Parameter: TimeDomain
Type: string, character vector
Values: "Continuous-time", "Discrete-time"
Default: "Continuous-time"

**Sample time (-1 for inherited) — Discrete interval between samples**
-1 (default) | positive scalar

Specify a sample time by entering a positive scalar value, such as 0.1. The default discrete sample time of -1 means that the block inherits its sample time from upstream blocks. However, it is recommended that you set the controller sample time explicitly, especially if you expect the sample time of upstream blocks to change. The effect of the controller coefficients P, I, D, and N depend on the sample time. Thus, for a given set of coefficient values, changing the sample time changes the performance of the controller.

See “Specify Sample Time” for more information.

To implement a continuous-time controller, set Time domain to Continuous-time.
**Tip** If you want to run the block with an externally specified or variable sample time, set this parameter to -1 and put the block in a Triggered Subsystem. Then, trigger the subsystem at the desired sample time.

**Dependencies**

To enable this parameter, set **Time domain** to **Discrete-time**.

**Programmatic Use**

**Block Parameter:** SampleTime

- **Type:** scalar
- **Values:** -1, positive scalar
- **Default:** -1

**Integrator method — Method for computing integral in discrete-time controller**

- **Forward Euler** (default) | **Backward Euler** | **Trapezoidal**

In discrete time, the integral term of the controller transfer function is $Ia(z)$, where $a(z)$ depends on the integrator method you specify with this parameter.

**Forward Euler**

- Forward rectangular (left-hand) approximation,

$$\alpha(z) = \frac{T s}{z-1}.$$  

This method is best for small sampling times, where the Nyquist limit is large compared to the bandwidth of the controller. For larger sampling times, the **Forward Euler** method can result in instability, even when discretizing a system that is stable in continuous time.

**Backward Euler**

- Backward rectangular (right-hand) approximation,

$$\alpha(z) = \frac{T s z}{z-1}.$$  

An advantage of the **Backward Euler** method is that discretizing a stable continuous-time system using this method always yields a stable discrete-time result.

**Trapezoidal**

- Bilinear approximation,
\[ \alpha(z) = \frac{T_s z + 1}{2 z - 1}. \]

An advantage of the Trapezoidal method is that discretizing a stable continuous-time system using this method always yields a stable discrete-time result. Of all available integration methods, the Trapezoidal method yields the closest match between frequency-domain properties of the discretized system and the corresponding continuous-time system.

**Tip** The controller formula for the current setting is displayed in the **Compensator formula** section of the block parameters and under the mask.

For more information about discrete-time integration, see the Discrete-Time Integrator block reference page.

**Dependencies**

To enable this parameter, set **Time Domain** to Discrete-time and set **Controller** to a controller type with integral action.

**Programmatic Use**

**Block Parameter:** IntegratorMethod  
**Type:** string, character vector  
**Values:** "Forward Euler", "Backward Euler", "Trapezoidal"  
**Default:** "Forward Euler"

**Filter method — Method for computing derivative in discrete-time controller**

Forward Euler (default) | Backward Euler | Trapezoidal

In discrete time, the derivative term of the controller transfer function is:

\[ D \left[ \frac{N}{1 + Na(z)} \right]. \]

where \( \alpha(z) \) depends on the filter method you specify with this parameter.

**Forward Euler**

Forward rectangular (left-hand) approximation,

\[ \alpha(z) = \frac{T_s}{z - 1}. \]
This method is best for small sampling times, where the Nyquist limit is large compared to the bandwidth of the controller. For larger sampling times, the Forward Euler method can result in instability, even when discretizing a system that is stable in continuous time.

**Backward Euler**

Backward rectangular (right-hand) approximation,

\[ \alpha(z) = \frac{T_s z}{z - 1}. \]

An advantage of the Backward Euler method is that discretizing a stable continuous-time system using this method always yields a stable discrete-time result.

**Trapezoidal**

Bilinear approximation,

\[ \alpha(z) = \frac{T_s z + 1}{2 z - 1}. \]

An advantage of the Trapezoidal method is that discretizing a stable continuous-time system using this method always yields a stable discrete-time result. Of all available integration methods, the Trapezoidal method yields the closest match between frequency-domain properties of the discretized system and the corresponding continuous-time system.

**Tip** The controller formula for the current setting is displayed in the **Compensator formula** section of the block parameters and under the mask.

For more information about discrete-time integration, see the Discrete-Time Integrator block reference page.

**Dependencies**

To enable this parameter, set **Time Domain** to Discrete-time and set **Controller** to a controller type with derivative action.

**Programmatic Use**

**Block Parameter**: FilterMethod  
**Type**: string, character vector  
**Values**: "Forward Euler", "Backward Euler", "Trapezoidal"
**Default:** "Forward Euler"

**Main**

**Source — Source for controller gains and filter coefficient**

- **internal** (default) | **external**

**internal**

Specify the controller gains, filter coefficient, and setpoint weights using the block parameters \( P, I, D, N, b, \) and \( c \) respectively.

**external**

Specify the PID gains, filter coefficient, and setpoint weights externally using block inputs. An additional input port appears on the block for each parameter that is required for the current controller type.

Enabling external inputs for the parameters allows you to compute their values externally to the block and provide them to the block as signal inputs.

External input is useful, for example, when you want to map a different PID parameterization to the PID gains of the block. You can also use external gain input to implement gain-scheduled PID control. In gain-scheduled control, you determine the PID gains by logic or other calculation in your model and feed them to the block.

When you supply gains externally, time variations in the integral and derivative gain values are integrated and differentiated, respectively. The derivative setpoint weight \( c \) is also differentiated. This result occurs because in both continuous time and discrete time, the gains are applied to the signal before integration or differentiation. For example, for a continuous-time PID controller with external inputs, the integrator term is implemented as shown in the following illustration.

\[
\begin{align*}
    u_i &= \int (r - y)I \, dt.
\end{align*}
\]

Within the block, the input signal \( u \) is multiplied by the externally supplied integrator gain, \( I \), before integration. This implementation yields:
Thus, the integrator gain is included in the integral. Similarly, in the derivative term of the block, multiplication by the derivative gain precedes the differentiation, which causes the derivative gain $D$ and the derivative setpoint weight $c$ to be differentiated.

**Programmatic Use**

**Block Parameter:** ControllerParametersSource  
**Type:** string, character vector  
**Values:** "internal", "external"  
**Default:** "internal"

**Proportional (P) — Proportional gain**  
1 (default) | scalar | vector

Specify a finite, real gain value for the proportional gain. When **Controller form** is:

- **Parallel** — Proportional action is independent of the integral and derivative actions. For example, for a continuous-time 2-DOF PID controller in parallel form, the controller output $u$ is:

$$u = P(br - y) + I\frac{1}{s}(r - y) + D\frac{N}{1 + N\frac{1}{s}}(cr - y),$$

where $r$ is the reference signal, $y$ is the measured plant output signal, and $b$ and $c$ are the setpoint weights.

For a discrete-time 2-DOF controller in parallel form, the controller output is:

$$u = P(br - y) + I\alpha(z)(r - y) + D\frac{N}{1 + N\beta(z)}(cr - y),$$

where the **Integrator method** and **Filter method** parameters determine $\alpha(z)$ and $\beta(z)$, respectively.

- **Ideal** — The proportional gain multiples the integral and derivative terms. For example, for a continuous-time 2-DOF PID controller in ideal form, the controller output is:

$$u = P\left[(br - y) + I\frac{1}{s}(r - y) + D\frac{N}{1 + N\frac{1}{s}}(cr - y)\right].$$

For a discrete-time 2-DOF PID controller in ideal form, the transfer function is:
\[ u = P \left[ (br - y) + I\alpha(z)(r - y) + D \frac{N}{1 + N\beta(z)} (cr - y) \right], \]

where the **Integrator method** and **Filter method** parameters determine \( \alpha(z) \) and \( \beta(z) \), respectively.

**Tunable:** Yes

**Dependencies**

To enable this parameter, set the Controller parameters **Source** to **internal**.

**Programmatic Use**

**Block Parameter:** \( P \)

**Type:** scalar, vector

**Default:** 1

**Integral (I) — Integral gain**

1 (default) | scalar | vector

Specify a finite, real gain value for the integral gain.

**Tunable:** Yes

**Dependencies**

To enable this parameter, in the **Main** tab, set the controller-parameters **Source** to **internal**, and set **Controller** to a type that has integral action.

**Programmatic Use**

**Block Parameter:** \( I \)

**Type:** scalar, vector

**Default:** 1

**Derivative (D) — Derivative gain**

0 (default) | scalar | vector

Specify a finite, real gain value for the derivative gain.

**Tunable:** Yes

**Dependencies**

To enable this parameter, in the **Main** tab, set the controller-parameters **Source** to **internal**, and set **Controller** to **PID** or **PD**.
**Programmatic Use**

**Block Parameter:** `D`

**Type:** scalar, vector

**Default:** 0

**Use filtered derivative — Apply filter to derivative term**

* on (default) | off *

For discrete-time PID controllers only, clear this option to replace the filtered derivative with an unfiltered discrete-time differentiator. When you do so, the derivative term of the controller output becomes:

\[ \frac{D}{1 - z^{-1}} \left( c_r - y \right). \]

For continuous-time PID controllers, the derivative term is always filtered.

**Dependencies**

To enable this parameter, set **Time domain** to **Discrete-time**, and set **Controller** to a type that has a derivative term.

**Programmatic Use**

**Block Parameter:** `UseFilter`

**Type:** string, character vector

**Values:** "on", "off"

**Default:** "on"

**Filter coefficient (N) — Derivative filter coefficient**

* 100 (default) | scalar | vector *

Specify a finite, real gain value for the filter coefficient. The filter coefficient determines the pole location of the filter in the derivative action of the block. The location of the filter pole depends on the **Time domain** parameter.

- When **Time domain** is **Continuous-time**, the pole location is \( s = -N \).
- When **Time domain** is **Discrete-time**, the pole location depends on the **Filter method** parameter.

<table>
<thead>
<tr>
<th>Filter Method</th>
<th>Location of Filter Pole</th>
</tr>
</thead>
<tbody>
<tr>
<td>Forward Euler</td>
<td>( z_{pole} = 1 - NT_s )</td>
</tr>
</tbody>
</table>
### Filter Method

<table>
<thead>
<tr>
<th>Filter Method</th>
<th>Location of Filter Pole</th>
</tr>
</thead>
<tbody>
<tr>
<td>Backward Euler</td>
<td>$z_{pole} = \frac{1}{1 + NT_s}$</td>
</tr>
<tr>
<td>Trapezoidal</td>
<td>$z_{pole} = \frac{1 - NT_s/2}{1 + NT_s/2}$</td>
</tr>
</tbody>
</table>

The block does not support $N = \infty$ (ideal unfiltered derivative). When the **Time domain** is Discrete-time, you can clear **Use filtered derivative** to remove the derivative filter.

**Tunable:** Yes

**Dependencies**

To enable this parameter, in the **Main** tab, set the controller-parameters **Source** to internal and set **Controller** to PID or PD.

**Programmatic Use**

**Block Parameter:** $N$

**Type:** scalar, vector

**Default:** 100

### Setpoint weight ($b$) — Proportional setpoint weight

1 (default) | scalar | vector

Setpoint weight on the proportional term of the controller. The proportional term of a 2-DOF controller output is $P(br-y)$, where $r$ is the reference signal and $y$ is the measured plant output. Setting $b$ to 0 eliminates proportional action on the reference signal, which can reduce overshoot in the system response to step changes in the setpoint. Changing the relative values of $b$ and $c$ changes the balance between disturbance rejection and setpoint tracking.

**Tunable:** Yes

**Dependencies**

To enable this parameter, in the **Main** tab, set the controller-parameters **Source** to internal.

**Programmatic Use**

**Block Parameter:** $b$
Type: scalar, vector
Default: 1

**Setpoint weight (c) — Derivative setpoint weight**

1 (default) | scalar | vector

Setpoint weight on the derivative term of the controller. The derivative term of a 2-DOF controller acts on \(cr-y\), where \(r\) is the reference signal and \(y\) is the measured plant output. Thus, setting \(c\) to 0 eliminates derivative action on the reference signal, which can reduce transient response to step changes in the setpoint. Setting \(c\) to 0 can yield a controller that achieves both effective disturbance rejection and smooth setpoint tracking without excessive transient response. Changing the relative values of \(b\) and \(c\) changes the balance between disturbance rejection and setpoint tracking.

Tunable: Yes

Dependencies

To enable this parameter, in the **Main** tab, set the controller-parameters **Source** to **internal** and set **Controller** to a type that has derivative action.

Programmatic Use

Block Parameter: c
Type: scalar, vector
Default: 1

**Select tuning method — Tool for automatic tuning of controller coefficients**

Transfer Function Based (PID Tuner App) (default) | Frequency Response Based

If you have Simulink Control Design software, you can automatically tune the PID coefficients when they are internal to the block. To do so, use this parameter to select a tuning tool, and click **Tune**.

Transfer Function Based (PID Tuner App)

Use **PID Tuner**, which lets you interactively tune PID coefficients while examining relevant system responses to validate performance. **PID Tuner** can tune all the coefficients \(P, I, D\), and \(N\), and the setpoint coefficients \(b\) and \(c\). By default, **PID Tuner** works with a linearization of your plant model. For models that cannot be linearized, you can tune PID coefficients against a plant model estimated from simulated or measured response data. For more information, see “Design Two-Degree-of-Freedom PID Controllers” (Simulink Control Design).
Frequency Response Based

Use **Frequency Response Based PID Tuner**, which tunes PID controller coefficients based on frequency-response estimation data obtained by simulation. This tuning approach is especially useful for plants that are not linearizable or that linearize to zero. **Frequency Response Based PID Tuner** tunes the coefficients $P$, $I$, $D$, and $N$, but does not tune the setpoint coefficients $b$ and $c$. For more information, see “Design PID Controller from Plant Frequency-Response Data” (Simulink Control Design).

Both of these tuning methods assume a single-loop control configuration. Simulink Control Design software includes other tuning approaches that suit more complex configurations. For information about other ways to tune a PID Controller block, see “Choose a Control Design Approach” (Simulink Control Design).

Dependencies

To enable this parameter, in the **Main** tab, set the controller-parameters **Source** to internal.

**Enable zero-crossing detection — Detect zero crossings on reset and on entering or leaving a saturation state**

on (default) | off

Zero-crossing detection can accurately locate signal discontinuities without resorting to excessively small time steps that can lead to lengthy simulation times. If you select **Limit output** or activate **External reset** in your PID Controller block, activating zero-crossing detection can reduce computation time in your simulation. Selecting this parameter activates zero-crossing detection:

- At initial-state reset
- When entering an upper or lower saturation state
- When leaving an upper or lower saturation state

For more information about zero-crossing detection, see “Zero-Crossing Detection”.

**Programmatic Use**

**Block Parameter:** **ZeroCross**

**Type:** string, character vector

**Values:** "on", "off"

**Default:** "on"
Initialization

Source — Source for integrator and derivative initial conditions

internal (default) | external

Simulink uses initial conditions to initialize the integrator and derivative-filter (or the unfiltered derivative) output at the start of a simulation or at a specified trigger event. (See the External reset parameter.) These initial conditions determine the initial block output. Use this parameter to select how to supply the initial condition values to the block.

internal

Specify the initial conditions using the Integrator Initial condition and Filter Initial condition parameters. If Use filtered derivative is not selected, use the Differentiator parameter to specify the initial condition for the unfiltered differentiator instead of a filter initial condition.

external

Specify the initial conditions externally using block inputs. Additional input ports I_o and D_o appear on the block. If Use filtered derivative is not selected, supply the initial condition for the unfiltered differentiator at D_o instead of a filter initial condition.

Programmatic Use

Block Parameter: InitialConditionSource

Type: string, character vector

Values: "internal", "external"

Default: "internal"

Integrator — Integrator initial condition

0 (default) | scalar | vector

Simulink uses the integrator initial condition to initialize the integrator at the start of a simulation or at a specified trigger event (see External reset). The integrator initial condition and the filter initial condition determine the initial output of the PID controller block.

The integrator initial condition cannot be NaN or Inf.

Dependencies

To use this parameter, in the Initialization tab, set Source to internal, and set Controller to a type that has integral action.
**Programmatic Use**

**Block Parameter:** InitialConditionForIntegrator  
**Type:** scalar, vector  
**Default:** 0

**Filter — Filter initial condition**  
0 (default) | scalar | vector

Simulink uses the filter initial condition to initialize the derivative filter at the start of a simulation or at a specified trigger event (see **External reset**). The integrator initial condition and the filter initial condition determine the initial output of the PID controller block.

The filter initial condition cannot be NaN or Inf.

**Dependencies**

To use this parameter, in the **Initialization** tab, set **Source** to internal, and use a controller that has a derivative filter.

**Programmatic Use**

**Block Parameter:** InitialConditionForFilter  
**Type:** scalar, vector  
**Default:** 0

**Differentiator — Initial condition for unfiltered derivative**  
0 (default) | scalar | vector

When you use an unfiltered derivative, Simulink uses this parameter to initialize the differentiator at the start of a simulation or at a specified trigger event (see **External reset**). The integrator initial condition and the derivative initial condition determine the initial output of the PID controller block.

The derivative initial condition cannot be NaN or Inf.

**Dependencies**

To use this parameter, set **Time domain** to Discrete-time, clear the **Use filtered derivative** check box, and in the **Initialization** tab, set **Source** to internal.

**Programmatic Use**

**Block Parameter:** DifferentiatorICPrevScaledInput  
**Type:** scalar, vector
Default: 0

**Initial condition setting — Location at which initial condition is applied**

State (most efficient) (default) | Output

Use this parameter to specify whether to apply the **Integrator Initial condition** and **Filter Initial condition** parameter to the corresponding block state or output. You can change this parameter at the command line only, using `set_param` to set the **InitialConditionSetting** parameter of the block.

**State (most efficient)**

Use this option in all situations except when the block is in a triggered subsystem or a function-call subsystem and simplified initialization mode is enabled.

**Output**

Use this option when the block is in a triggered subsystem or a function-call subsystem and simplified initialization mode is enabled.

For more information about the **Initial condition setting** parameter, see the Discrete-Time Integrator block.

This parameter is only accessible through programmatic use.

**Programmatic Use**

**Block Parameter:** InitialConditionSetting

**Type:** string, character vector

**Values:** "state", "output"

**Default:** "state"

**External reset — Trigger for resetting integrator and filter values**

none (default) | rising | falling | either | level

Specify the trigger condition that causes the block to reset the integrator and filter to initial conditions. (If **Use filtered derivative** is not selected, the trigger resets the integrator and differentiator to initial conditions.) Selecting any option other than none enables the **Reset** port on the block for the external reset signal.

**none**

The integrator and filter (or differentiator) outputs are set to initial conditions at the beginning of simulation, and are not reset during simulation.

**rising**

Reset the outputs when the reset signal has a rising edge.
falling
  Reset the outputs when the reset signal has a falling edge.
either
  Reset the outputs when the reset signal either rises or falls.
level
  Reset the outputs when the reset signal either:
  • Is nonzero at the current time step
  • Changes from nonzero at the previous time step to zero at the current time step

  This option holds the outputs to the initial conditions while the reset signal is nonzero.

Dependencies

To enable this parameter, set Controller to a type that has derivative or integral action.

Programmatic Use

Block Parameter: ExternalReset
Type: string, character vector
Values: "none", "rising", "falling", "either", "level"
Default: "none"

Ignore reset when linearizing — Force linearization to ignore reset

off (default) | on

Select to force Simulink and Simulink Control Design linearization commands to ignore any reset mechanism specified in the External reset parameter. Ignoring reset states allows you to linearize a model around an operating point even if that operating point causes the block to reset.

Programmatic Use

Block Parameter: IgnoreLimit
Type: string, character vector
Values: "off", "on"
Default: "off"

Enable tracking mode — Activate signal tracking

off (default) | on

Signal tracking lets the block output follow a tracking signal that you provide at the TR port. When signal tracking is active, the difference between the tracking signal and the
block output is fed back to the integrator input with a gain $K_t$, specified by the Tracking gain ($K_t$) parameter. Signal tracking has several applications, including bumpless control transfer and avoiding windup in multiloop control structures.

**Bumpless control transfer**

Use signal tracking to achieve bumpless control transfer in systems that switch between two controllers. Suppose you want to transfer control between a PID controller and another controller. To do so, connecting the controller output to the TR input as shown in the following illustration.

For more information, see “Bumpless Control Transfer with a Two-Degree-of-Freedom PID Controller” on page 14-129.

**Multiloop control**

Use signal tracking to prevent block windup in multiloop control approaches. For an example illustrating this approach with a 1DOF PID controller, see “Prevent Block Windup in Multiloop Control” on page 14-127.

**Dependencies**

To enable this parameter, set **Controller** to a type that has integral action.
Programmatic Use
Block Parameter: TrackingMode
Type: string, character vector
Values: "off", "on"
Default: "off"

**Tracking coefficient (Kt) — Gain of signal-tracking feedback loop**

1 (default) | scalar

When you select Enable tracking mode, the difference between the signal TR and the block output is fed back to the integrator input with a gain Kt. Use this parameter to specify the gain in that feedback loop.

Dependencies

To enable this parameter, select Enable tracking mode.

---

Programmatic Use
Block Parameter: Kt
Type: scalar
Default: 1

---

**Output saturation**

**Limit Output — Limit block output to specified saturation values**

off (default) | on

Activating this option limits the block output internally to the block, so that you do not need a separate Saturation on page 1-1704 block after the controller. It also allows you to activate the anti-windup mechanism built into the block (see the Anti-windup method parameter). Specify the saturation limits using the **Lower saturation limit** and **Upper saturation limit** parameters.

Programmatic Use
Block Parameter: LimitOutput
Type: string, character vector
Values: "off", "on"
Default: "off"

**Upper limit — Upper saturation limit for block output**

Inf (default) | scalar
Specify the upper limit for the block output. The block output is held at the **Upper saturation limit** whenever the weighted sum of the proportional, integral, and derivative actions exceeds that value.

**Dependencies**

To enable this parameter, select **Limit output**.

**Programmatic Use**

**Block Parameter:** UpperSaturationLimit  
**Type:** scalar  
**Default:** Inf

**Lower limit — Lower saturation limit for block output**

-Inf (default) | scalar

Specify the lower limit for the block output. The block output is held at the **Lower saturation limit** whenever the weighted sum of the proportional, integral, and derivative actions goes below that value.

**Dependencies**

To enable this parameter, select **Limit output**.

**Programmatic Use**

**Block Parameter:** LowerSaturationLimit  
**Type:** scalar  
**Default:** -Inf

**Ignore saturation when linearizing — Force linearization to ignore output limits**

off (default) | on

Force Simulink and Simulink Control Design linearization commands to ignore block output limits specified in the **Upper limit** and **Lower limit** parameters. Ignoring output limits allows you to linearize a model around an operating point even if that operating point causes the block to exceed the output limits.

**Dependencies**

To enable this parameter, select the **Limit output** parameter.

**Programmatic Use**

**Block Parameter:** LinearizeAsGain
Type: string, character vector
Values: "off", "on"
Default: "off"

Anti-windup method — Integrator anti-windup method
none (default) | back-calculation | clamping

When you select Limit output and the weighted sum of the controller components exceeds the specified output limits, the block output holds at the specified limit. However, the integrator output can continue to grow (integrator windup), increasing the difference between the block output and the sum of the block components. In other words, the internal signals in the block can be unbounded even if the output appears bounded by saturation limits. Without a mechanism to prevent integrator windup, two results are possible:

- If the sign of the signal entering the integrator never changes, the integrator continues to integrate until it overflows. The overflow value is the maximum or minimum value for the data type of the integrator output.
- If the sign of the signal entering the integrator changes once the weighted sum has grown beyond the output limits, it can take a long time to unwind the integrator and return the weighted sum within the block saturation limit.

In either case, controller performance can suffer. To combat the effects of windup without an anti-windup mechanism, it may be necessary to detune the controller (for example, by reducing the controller gains), resulting in a sluggish controller. To avoid this problem, activate an anti-windup mechanism using this parameter.

none

Do not use an anti-windup mechanism.

back-calculation

Unwind the integrator when the block output saturates by feeding back to the integrator the difference between the saturated and unsaturated control signal. The following diagram represents the back-calculation feedback circuit for a continuous-time controller. To see the actual feedback circuit for your controller configuration, right-click on the block and select Mask > Look Under Mask.
Use the **Back-calculation coefficient (Kb)** parameter to specify the gain of the anti-windup feedback circuit. It is usually satisfactory to set $K_b = I$, or for controllers with derivative action, $K_b = \sqrt{I \cdot D}$. Back-calculation can be effective for plants with relatively large dead time [1].

**Clamping**
Integration stops when the sum of the block components exceeds the output limits and the integrator output and block input have the same sign. Integration resumes when the sum of the block components exceeds the output limits and the integrator output and block input have opposite sign. Clamping is sometimes referred to as conditional integration.

Clamping can be useful for plants with relatively small dead times, but can yield a poor transient response for large dead times [1].

**Dependencies**
To enable this parameter, select the **Limit output** parameter.

**Programmatic Use**
**Block Parameter:** AntiWindupMode  
**Type:** string, character vector  
**Values:** "none","back-calculation","clamping"
**Default:** "none"

**Back-calculation coefficient (Kb) — Gain coefficient of anti-windup feedback loop**  
1 (default) | scalar

The back-calculation anti-windup method unwinds the integrator when the block output saturates. It does so by feeding back to the integrator the difference between the saturated and unsaturated control signal. Use the **Back-calculation coefficient (Kb)** parameter to specify the gain of the anti-windup feedback circuit. For more information, see the **Anti-windup method** parameter.

**Dependencies**

To enable this parameter, select the **Limit output** parameter, and set the **Anti-windup method** parameter to **back-calculation**.

**Programmatic Use**

**Block Parameter:** Kb  
**Type:** scalar  
**Default:** 1

**Data Types**

The parameters in this tab are primarily of use in fixed-point code generation using Fixed-Point Designer. They define how numeric quantities associated with the block are stored and processed when you generate code.

If you need to configure data types for fixed-point code generation, click **Open Fixed-Point Tool** and use that tool to configure the rest of the parameters in the tab. For information about using Fixed-Point Tool, see “Autoscaling Data Objects Using the Fixed-Point Tool” (Fixed-Point Designer).

After you use Fixed-Point Tool, you can use the parameters in this tab to make adjustments to fixed-point data-type settings if necessary. For each quantity associated with the block, you can specify:

- Floating-point or fixed-point data type, including whether the data type is inherited from upstream values in the block.
- The minimum and maximum values for the quantity, which determine how the quantity is scaled for fixed-point representation.
The specific quantities listed in the Data Types tab vary depending on how you configure the PID controller block. In general, you can configure data types for the following types of quantities:

- **Product output** — Stores the result of a multiplication carried out under the block mask. For example, \textbf{P product output} stores the output of the gain block that multiplies the block input with the proportional gain \( P \).
- **Parameter** — Stores the value of a numeric block parameter, such as \( P \), \( I \), or \( D \).
- **Block output** — Stores the output of a block that resides under the PID controller block mask. For example, use \textbf{Integrator output} to specify the data type of the
output of the block called Integrator. This block resides under the mask in the Integrator subsystem, and computes integrator term of the controller action.

- Accumulator — Stores values associated with a sum block. For example, **SumI2 Accumulator** sets the data type of the accumulator associated with the sum block SumI2. This block resides under the mask in the Back Calculation subsystem of the Anti-Windup subsystem.

In general, you can find the block associated with any listed parameter by looking under the PID Controller block mask and examining its subsystems. You can also use the Model Explorer to search under the mask for the listed parameter name, such as SumI2. (See “Search and Edit Using Model Explorer”.)

**Matching Input and Internal Data Types**

By default, all data types in the block are set to **Inherit: Inherit via internal rule**. With this setting, Simulink chooses data types to balance numerical accuracy, performance, and generated code size, while accounting for the properties of the embedded target hardware.

Under some conditions, incompatibility can occur between data types within the block. For instance, in continuous time, the Integrator block under the mask can accept only signals of type *double*. If the block input signal is a type that cannot be converted to *double*, such as *uint16*, the internal rules for type inheritance generate an error when you generate code.

To avoid such errors, you can use the Data Types settings to force a data type conversion. For instance, you can explicitly set **P product output**, **I product output**, and **D product output** to *double*, ensuring that the signals reaching the continuous-time integrators are of type *double*.

In general, it is not recommended to use the block in continuous time for code generation applications. However, similar data type errors can occur in discrete time, if you explicitly set some values to data types that are incompatible with downstream signal constraints within the block. In such cases, use the Data Types settings to ensure that all data types are internally compatible.

**Fixed-Point Operational Parameters**

**Integer rounding mode** — Rounding mode for fixed-point operations
- Floor (default)  
- Ceiling  
- Convergent  
- Nearest  
- Round  
- Simplest  
- Zero
Specify the rounding mode for fixed-point operations. For more information, see “Rounding” (Fixed-Point Designer).

Block parameters always round to the nearest representable value. To control the rounding of a block parameter, enter an expression using a MATLAB rounding function into the mask field.

**Programmatic Use**

**Block Parameter:** RndMeth  
**Type:** character vector  
**Values:** 'Ceiling' | 'Convergent' | 'Floor' | 'Nearest' | 'Round' | 'Simplest' | 'Zero'  
**Default:** 'Floor'

**Saturate on integer overflow — Method of overflow action**

*off (default) | on*

Specify whether overflows saturate or wrap.

- **off** — Overflows wrap to the appropriate value that the data type can represent.
  
  For example, the number 130 does not fit in a signed 8-bit integer and wraps to -126.

- **on** — Overflows saturate to either the minimum or maximum value that the data type can represent.

  For example, an overflow associated with a signed 8-bit integer can saturate to -128 or 127.

**Tip**

- Consider selecting this check box when your model has a possible overflow and you want explicit saturation protection in the generated code.
- Consider clearing this check box when you want to optimize efficiency of your generated code.

Clearing this check box also helps you to avoid overspecifying how a block handles out-of-range signals. For more information, see “Check for Signal Range Errors”.

- When you select this check box, saturation applies to every internal operation on the block, not just the output or result.
- In general, the code generation process can detect when overflow is not possible. In this case, the code generator does not produce saturation code.
**Programmatic Use**

**Block Parameter:** SaturateOnIntegerOverflow  
**Type:** character vector  
**Values:** 'off' | 'on'  
**Default:** 'off'

*Lock data type settings against changes by the fixed-point tools — Prevent fixed-point tools from overriding data types*

*off* (default) | *on*

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on this block. For more information, see “Lock the Output Data Type Setting” (Fixed-Point Designer).

**Programmatic Use**

**Block Parameter:** LockScale  
**Type:** character vector  
**Values:** 'off' | 'on'  
**Default:** 'off'

**State Attributes**

The parameters in this tab are primarily of use in code generation.

**State name (e.g., 'position') — Name for continuous-time filter and integrator states**

' ' (default) | character vector

Assign a unique name to the state associated with the integrator or the filter, for continuous-time PID controllers. (For information about state names in a discrete-time PID controller, see the **State name** parameter.) The state name is used, for example:

- For the corresponding variable in generated code  
- As part of the storage name when logging states during simulation  
- For the corresponding state in a linear model obtained by linearizing the block

A valid state name begins with an alphabetic or underscore character, followed by alphanumeric or underscore characters.

**Dependencies**

To enable this parameter, set **Time domain** to **Continuous-time**.
**Programmatic Use**

Parameter: IntegratorContinuousStateAttributes, FilterContinuousStateAttributes  
Type: character vector  
Default: ''

**State name — Names for discrete-time filter and integrator states**

empty string (default) | string | character vector

Assign a unique name to the state associated with the integrator or the filter, for discrete-time PID controllers. (For information about state names in a continuous-time PID controller, see the State name (e.g., 'position') parameter.)

A valid state name begins with an alphabetic or underscore character, followed by alphanumeric or underscore characters. The state name is used, for example:

- For the corresponding variable in generated code
- As part of the storage name when logging states during simulation
- For the corresponding state in a linear model obtain by linearizing the block

For more information about the use of state names in code generation, see “Apply Storage Classes to Individual Signal, State, and Parameter Data Elements” (Simulink Coder).

**Dependencies**

To enable this parameter, set Time domain to Discrete-time.

**Programmatic Use**

Parameter: IntegratorStateIdentifier, FilterStateIdentifier  
Type: string, character vector  
Default: ""

**State name must resolve to Simulink signal object — Require that state name resolve to a signal object**

off (default) | on

Select this parameter to require that the discrete-time integrator or filter state name resolves to a Simulink signal object.

**Dependencies**

To enable this parameter for the discrete-time integrator or filter state:
1. Set **Time domain** to **Discrete-time**.
2. Specify a value for the integrator or filter **State name**.
3. Set the model configuration parameter **Signal resolution** to a value other than None.

Selecting this check box disables **Code generation storage class** for the corresponding integrator or filter state.

**Programmatic Use**

**Block Parameter:** IntegratorStateMustResolveToSignalObject, FilterStateMustResolveToSignalObject

**Type:** string, character vector

**Values:** "off", "on"

**Default:** "off"

**Code generation storage class — Storage class for code generation**

Auto (default) | ExportedGlobal | ImportedExtern | ImportedExternPointer

Select state storage class for code generation. If you do not need to interface to external code, select Auto.

For more information, see “Apply Storage Classes to Individual Signal, State, and Parameter Data Elements” (Simulink Coder) and “Apply Built-In and Customized Storage Classes to Data Elements” (Embedded Coder).

**Dependencies**

To enable this parameter for the discrete-time integrator or filter state:

1. Set **Time domain** to **Discrete-time**.
2. Specify a value for the integrator or filter **State name**.
3. Set the model configuration parameter **Signal resolution** to a value other than None.

**Programmatic Use**

**Block Parameter:** IntegratorRTWStateStorageClass, FilterRTWStateStorageClass

**Type:** string, character vector

**Values:** "Auto", "ExportedGlobal", "ImportedExtern" | "ImportedExternPointer"

**Default:** "Auto"
**Code generation storage type qualifier — Storage type qualifier**

empty string (default) | character vector | "const" | "volatile" | ...

Specify a storage type qualifier such as const or volatile.

**Note** This parameter will be removed in a future release. To apply storage type qualifiers to data, use custom storage classes and memory sections. Unless you use an ERT-based code generation target with Embedded Coder, custom storage classes and memory sections do not affect the generated code.

**Dependencies**

To enable this parameter, set *Code generation storage class* to any value other than Auto.

**Programmatic Use**

**Block Parameter:**

IntegratorRTWStateStorageTypeQualifier, FilterRTWStateStorageTypeQualifier

**Type:** string, character vector

**Values:** ","const","volatile"

**Default:** "."

**Block Characteristics**

<table>
<thead>
<tr>
<th>Data Types</th>
<th>double</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
More About

Decomposition of 2-DOF PID Controllers

A 2-DOF PID controller can be interpreted as a PID controller with a prefilter, or a PID controller with a feedforward element.

Prefilter Decomposition

In parallel form, a two-degree-of-freedom PID controller can be equivalently modeled by the following block diagram, where $C$ is a single degree-of-freedom PID controller and $F$ is a prefilter on the reference signal.

![Block diagram of a 2-DOF PID controller]

$Ref$ is the reference signal, $y$ is the feedback from the measured system output, and $u$ is the controller output. For a continuous-time 2-DOF PID controller in parallel form, the transfer functions for $F$ and $C$ are

$$F_{par}(s) = \frac{(bP + cDN)s^2 + (bPN + I)s + IN}{(P + DN)s^2 + (PN + I)s + IN},$$

$$C_{par}(s) = \frac{(P + DN)s^2 + (PN + I)s + IN}{s(s + N)},$$

where $b$ and $c$ are the setpoint weights.

For a 2-DOF PID controller in ideal form, the transfer functions are

$$F_{id}(s) = \frac{(b + cDN)s^2 + (bN + I)s + IN}{(1 + DN)s^2 + (N + I)s + IN},$$

$$C_{id}(s) = \frac{P(1 + DN)s^2 + (N + I)s + IN}{s(s + N)}.$$
A similar decomposition applies for a discrete-time 2-DOF controller.

**Feedforward Decomposition**

Alternatively, the parallel two-degree-of-freedom PID controller can be modeled by the following block diagram.

In this realization, $Q$ acts as feed-forward conditioning on the reference signal. For a continuous-time 2-DOF PID controller in parallel form, the transfer function for $Q$ is

$$Q_{par}(s) = \frac{(b - 1)P + (c - 1)DNs + (b - 1)PN}{s + N}.$$  

For a 2-DOF PID controller in ideal form, the transfer function is

$$Q_{id}(s) = P\frac{(b - 1) + (c - 1)DNs + (b - 1)N}{s + N}.$$  

The transfer functions for $C$ are the same as in the filter decomposition.

A similar decomposition applies for a discrete-time 2-DOF controller.

**References**

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

For continuous-time PID controllers (Time domain set to Continuous-time):

- Consider using “Model Discretizer” to map continuous-time blocks to discrete equivalents that support code generation. To access Model Discretizer, in the Apps tab, under Control Systems, click Model Discretizer.
- Not recommended for production code.

For discrete-time PID controllers (Time domain set to Discrete-time):

- Depends on absolute time when placed inside a triggered subsystem hierarchy.
- Generated code relies on memcpy or memset functions (string.h) under certain conditions.

PLC Code Generation
Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

Fixed-point code generation is supported for discrete-time PID controllers only (Time domain set to Discrete-time).

See Also
Derivative | Discrete PID Controller (2DOF) | Gain | Integrator | PID Controller

Introduced in R2009b
Polynomial

Perform evaluation of polynomial coefficients on input values

**Library:** Simulink / Math Operations

**Description**

The Polynomial block evaluates $P(u)$ at each time step for the input $u$. You define a set of polynomial coefficients in the form that the MATLAB `polyval` command accepts.

**Ports**

**Input**

**Port_1 — Input signal**
real scalar or vector

Value at which to evaluate the polynomial $P(u)$.

Data Types: single | double

**Output**

**Port_1 — Evaluated polynomial value**
real scalar or vector

Value of the polynomial $P(u)$ evaluated at the input signal.

Data Types: single | double
Parameters

Polynomial coefficients — Coefficients of polynomial to be evaluated
[ +2.081618890e-019, -1.441693666e-014, +4.719686976e-010, -8.536869453e-006, +1.621573104e-001, -8.087801117e+001 ] (default) | real array

Specify polynomial coefficients in MATLAB polyval form. The first coefficient corresponds to \( x^N \) and the remaining coefficients correspond to decreasing orders of \( x \). The last coefficient represents the constant for the polynomial.

Programmatic Use
Block Parameter: coefs
Type: real array
Default: [ +2.081618890e-019, -1.441693666e-014, +4.719686976e-010, -8.536869453e-006, +1.621573104e-001, -8.087801117e+001 ]

Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>double</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
</tr>
</tbody>
</table>

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.
PLC Code Generation
Generate Structured Text code using Simulink® PLC Coder™.

See Also

Topics
polyval

Introduced before R2006a
Prelookup

Compute index and fraction for Interpolation Using Prelookup block

Library:
- Simulink / Lookup Tables
- HDL Coder / Lookup Tables

Description

The Prelookup block calculates the index and interval fraction that specify how its input value \( u \) relates to the breakpoint dataset. The Prelookup block works best with the Interpolation Using Prelookup block. Feed the resulting index and fraction values into an Interpolation Using Prelookup block to interpolate an \( n \)-dimensional table. These two blocks have distributed algorithms. When combined together, they perform the same operation as the integrated algorithm in the n-D Lookup Table block. However, the Prelookup and Interpolation Using Prelookup blocks offer greater flexibility and more efficient simulation and code generation than the n-D Lookup Table block. For more information, see “Efficiency of Performance”.

Supported Block Operations

To use the Prelookup block, you must specify a set of breakpoint values. You choose whether to specify the breakpoint values directly on the dialog box or by feeding the values to a bp input port by setting the Source parameter to Dialog or Input port. Typically, this breakpoint data set corresponds to one dimension of the table data in an Interpolation Using Prelookup block. The Prelookup block generates a pair of outputs for each input value \( u \) by calculating:

- The index of the breakpoint set element that is less than or equal to \( u \) and forms an interval containing \( u \)
- The interval fraction in the range \( 0 \leq f < 1 \), representing the normalized position of \( u \) on the breakpoint interval between the index and the next index value for in-range input
For example, if the breakpoint data set is \([0 \ 5 \ 10 \ 20 \ 50 \ 100]\) and the input value \(u\) is 55, the index is 4 and the fractional value is 0.1. Labels for the index and interval fraction appear as \(k\) and \(f\) on the Prelookup block icon. The index value is zero based.

The interval fraction can be negative or greater than 1 for out-of-range input. See the Extrapolation method block parameter for more information.

**Ports**

**Input**

**Port_1** — Input signal, \(u\)
scalar | vector | matrix

The Prelookup block accepts real-valued signals of any numeric data type that Simulink supports, except Boolean. The Prelookup block supports fixed-point data types for signals and breakpoint data.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point | enumerated | bus

**bp** — Breakpoint data
1-D vector of real-valued signals

The Prelookup block accepts real-valued signals as breakpoint data of any numeric data type that Simulink supports, except Boolean.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point | enumerated

**Output**

**k** — Index of the interval containing the input, \(u\)
scalar | vector | matrix

The zero-based index, \(k\), is a real-valued integer that specifies the interval containing the input, \(u\).

**Dependencies**

To enable this port, set the Output selection to Index and fraction or Index only.
Data Types: int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

\( \text{f} \) — Fraction representing the normalized position of the input, \( u \), within the interval, \( k \)
scalar | vector | matrix

Fraction, \( f \), represents the normalized position of the input, \( u \), within the interval \( k \).

Dependencies

To enable this port, set the Output selection to Index and fraction.

Data Types: single | double | fixed point

\( \text{Port_2} \) — Bus containing index, \( k \), and fraction, \( f \)
bus

Outputting the index, \( k \), and fraction \( f \), as a bus object can help simplify the model.

Dependencies

To enable this port, set the Output selection to Index and fraction as bus.

Data Types: bus

Parameters

Main

Breakpoints data

Specification — Choose how to enter breakpoint data
Explicit values (default) | Even spacing | Breakpoint object

If you set this parameter to:

- Explicit values, the Source and Value parameters are visible on the dialog box.
- Even spacing, the First point, Spacing, and Number of points parameters are visible on the dialog box.
- Breakpoint object, the Name parameter is visible on the dialog box.
Programmatic Use
Block Parameter: BreakpointsSpecification
Type: character vector
Values: 'Explicit values' | 'Even spacing' | 'Breakpoint object'
Default: 'Explicit values'

Source — Specify source of breakpoint data
Dialog (default) | Input port

If you set Source to:

• Dialog, specify breakpoint data under Value.
• Input port, verify that an upstream signal supplies breakpoint data to the bp input port. Each breakpoint data set must be a strictly monotonically increasing vector that contains two or more elements. For this option, your block inherits breakpoint attributes from the bp input port.

Dependencies
To enable this parameter, set Specification to Explicit values.

Programmatic Use
Block Parameter: BreakpointsDataSource
Type: character vector
Values: 'Dialog' | 'Input port'
Default: 'Dialog'

Value — Breakpoint data values
[10:10:110] (default) | vector of strictly monotonically increasing values containing two or more elements

Explicitly specify the breakpoint data. Each breakpoint data set must be a strictly monotonically increasing vector that contains two or more elements. For this option, you specify additional breakpoint attributes on the Data Types pane.

To open the Lookup Table Editor, click Edit (see “Edit Lookup Tables”).

Note When you set Specification to Explicit values and Source to Input port, verify that an upstream signal supplies breakpoint data to the bp input port. Each breakpoint data set must be a strictly monotonically increasing vector that contains two or more elements. For this option, your block inherits breakpoint attributes (including data type) from the bp input port.
Dependencies
To enable this parameter, set **Specification** to **Explicit** values and **Source** to Dialog.

**Programmatic Use**
**Block Parameter:** BreakpointsData
**Type:** character vector
**Values:** vector of strictly monotonically increasing values containing two or more elements
**Default:** '10:10:110'

**First point** — First point in evenly spaced breakpoint data
10 (default) | real-valued scalar

**Dependencies**
To enable this parameter, set **Specification** to **Even** spacing.

**Programmatic Use**
**Block Parameter:** BreakpointsFirstPoint
**Type:** character vector
**Values:** real-valued scalar
**Default:** '10'

**Spacing** — Spacing between evenly spaced breakpoints
10 (default) | real-valued, positive scalar

**Dependencies**
To enable this parameter, set **Specification** to **Even** spacing.

**Programmatic Use**
**Block Parameter:** BreakpointsSpacing
**Type:** character vector
**Values:** real-valued, positive scalar
**Default:** '10'

**Number of points** — Number of evenly spaced points
11 (default) | real-valued, positive scalar

**Dependencies**
To enable this parameter, set **Specification** to **Even** spacing.
Programmatic Use
Block Parameter: BreakpointsNumPoints
Type: character vector
Values: real-valued, positive scalar
Default: '11'

Name — Name of a Simulink.Breakpoint object

Specify the name of a Simulink.Breakpoint object. A breakpoint object references Simulink breakpoint objects. If a Simulink.Breakpoint object does not exist, click the action button and select Create. The corresponding parameters of the new breakpoint object are automatically populated with the block information.

Dependencies
To enable this parameter, set Specification to Breakpoint object.

Programmatic Use
Block Parameter: BreakpointObject
Type: character vector
Values: Simulink.Breakpoint object
Default: ''

Algorithm

Output selection — Specify the signals the block outputs

Index and fraction (default)| Index and fraction as bus | Index only

If you want the block to output the index and interval fraction, you can specify whether the block outputs individual signals or a bus signal that includes both the index and fraction signals.

- Index only outputs just the index, without the fraction. Typical applications for this option include:
  - Feeding a Direct Lookup Table (n-D) block, with no interpolation on the interval
  - Feeding selection ports of a subtable selection for an Interpolation Using Prelookup block
  - Performing nonlinear quantizations
- **Index and fraction** outputs the index and fraction as individual signals.
- **Index and fraction as bus** outputs a bus signal that includes the index and fraction signals. Using a bus for these signals:
  - Simplifies the model by tying these two related signals together
  - Creates a testpoint `DpResult` structure for the AUTOSAR 4.0 library
  - For the AUTOSAR 4.0 library, avoids the creation of extra copies during code generation when the Prelookup and Interpolation Using Prelookup blocks are in separate models

**Note** Selecting **Index and fraction as bus** displays the **Output** parameter in the **Data Types** pane and sets the **Output** parameter to **Inherit: auto**. Change this default value to specify a user-defined bus object. For details about defining the bus object, see the **Output** parameter description.

**Programmatic Use**

**Block Parameter:** `OutputSelection`  
**Type:** character vector  
**Values:** 'Index and fraction' | 'Index and fraction as bus' | 'Index only'  
**Default:** 'Index and fraction'

**Index search method — Method for searching breakpoint data**

Binary search (default) | Linear search | Evenly spaced points

Each search method has speed advantages in different situations:

- For evenly spaced breakpoint sets (for example, 10, 20, 30, and so on), you achieve optimal speed by selecting Evenly spaced points to calculate table indices. This algorithm uses only the first two breakpoints of a set to determine the offset and spacing of the remaining points.
- For unevenly spaced breakpoint sets, follow these guidelines:
  - If input values for \( u \) do not vary significantly between time steps, selecting Linear search with **Begin index search using previous index result** produces the best performance.
  - If input values for \( u \) jump more than one or two table intervals per time step, selecting Binary search produces the best performance.
A suboptimal choice of index search method can lead to slow performance of models that rely heavily on lookup tables.

**Note** The generated code stores only the first breakpoint, the spacing, and the number of breakpoints when:

- The breakpoint data is not tunable.
- The index search method is *Evenly spaced points*.

---

**Programmatic Use**

**Block Parameter:** IndexSearchMethod  
**Type:** character vector  
**Values:** 'Evenly spaced points' | 'Linear search' | 'Binary search'  
**Default:** 'Binary search'

**Begin index search using previous index result — Start search using the index found at the previous time step**

- off (default)  
- on

For input values of $u$ that change slowly with respect to the interval size, enabling this option can improve performance. Otherwise, the linear search and binary search methods can take longer, especially for large breakpoint sets.

**Programmatic Use**

**Block Parameter:** BeginIndexSearchUsingPreviousIndexResult  
**Type:** character vector  
**Values:** 'off' | 'on'  
**Default:** 'off'

**Extrapolation method — Method for handling out-of-range input values**

- Clip (default)  
- Linear

Options include:

- Clip
<table>
<thead>
<tr>
<th>Block Input</th>
<th>Block Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Less than the first breakpoint</td>
<td>• Index of the first breakpoint (for example, 0)</td>
</tr>
<tr>
<td></td>
<td>• Interval fraction of 0</td>
</tr>
<tr>
<td>Greater than the last breakpoint</td>
<td>• Index of the next-to-last breakpoint</td>
</tr>
<tr>
<td></td>
<td>• Interval fraction of 1</td>
</tr>
</tbody>
</table>

Suppose the range is \([1 \ 2 \ 3]\) and you select this option. If \(u = 0.5\), the index is 0 and the interval fraction is 0. If \(u = 3.5\), the index is 1 and the interval fraction is 1.

- Linear

<table>
<thead>
<tr>
<th>Block Input</th>
<th>Block Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Less than the first breakpoint</td>
<td>• Index of the first breakpoint (for example, 0)</td>
</tr>
<tr>
<td></td>
<td>• Interval fraction that represents the linear distance from (u) to the first breakpoint</td>
</tr>
<tr>
<td>Greater than the last breakpoint</td>
<td>• Index of the next-to-last breakpoint</td>
</tr>
<tr>
<td></td>
<td>• Interval fraction that represents the linear distance from the next-to-last breakpoint to (u)</td>
</tr>
</tbody>
</table>

Suppose the range is \([1 \ 2 \ 3]\) and you select this option. If \(u = 0.5\), the index is 0 and the interval fraction is -0.5. If \(u = 3.5\), the index is 1 and the interval fraction is 1.5.

**Note** The Prelookup block supports linear extrapolation only when all of the following conditions are true:

- The input \(u\), breakpoint data, and fraction output use floating-point data types.
- The index uses a built-in integer data type.

**Programmatic Use**

**Block Parameter:** ExtrapMethod
**Type:** character vector  
**Values:** 'Clip' | 'Linear'  
**Default:** 'Clip'

**Use last breakpoint for input at or above upper limit — Method of handling inputs at or above upper limit**  
*off* (default) | *on*

Specify how to index input values of u that are greater than or equal to the last breakpoint. The index value is zero based. When input equals the last breakpoint, block outputs differ as follows.

<table>
<thead>
<tr>
<th>Check Box</th>
<th>Block Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selected (on)</td>
<td>• Index of the last element in the breakpoint data set</td>
</tr>
<tr>
<td></td>
<td>• Interval fraction of 0</td>
</tr>
<tr>
<td>Cleared (off)</td>
<td>• Index of the next-to-last breakpoint</td>
</tr>
<tr>
<td></td>
<td>• Interval fraction of 1</td>
</tr>
</tbody>
</table>

**Tip** When you select **Use last breakpoint for input at or above upper limit** for a Prelookup block, you must also select **Valid index input may reach last index** for the Interpolation Using Prelookup block to which it connects. This action allows the blocks to use the same indexing convention when accessing the last elements of their breakpoint and table data sets.

**Dependencies**

This check box is visible only when:

- **Output only the index** is cleared  
- **Extrapolation method** is Clip

However, when **Output only the index** is selected and **Extrapolation method** is Clip, the block behaves as if this check box is selected, even though it is invisible.

**Programmatic Use**

**Block Parameter:** UseLastBreakpoint  
**Type:** character vector  
**Values:** 'off' | 'on'
**Default:** 'off'

**Diagnostic for out-of-range input — Block action when input is out of range**

None (default) | Warning | Error

Options include:

- None — Produce no response.
- Warning — Display a warning and continue the simulation.
- Error — Terminate the simulation and display an error.

**Programmatic Use**

**Block Parameter:** DiagnosticForOutOfRangeInput  
Type: character vector  
**Values:** 'None' | 'Warning' | 'Error'  
**Default:** 'None'

**Code generation**

**Remove protection against out-of-range input in generated code — Remove code that checks for out-of-range breakpoint inputs**

Off (default) | On

<table>
<thead>
<tr>
<th>Check Box</th>
<th>Result</th>
<th>When to Use</th>
</tr>
</thead>
</table>
| **On**    | Generated code does not include conditional statements to check for out-of-range breakpoint inputs.  
When the input k or f is out-of-range, it may cause undefined behavior for generated code and simulations using accelerator mode. | For code efficiency |
| **Off**   | Generated code includes conditional statements to check for out-of-range breakpoint inputs. | For safety-critical applications |
If your input is not out-of-range, you can select the **Remove protection against out-of-range index in generated code** check box for code efficiency. By default, this check box is cleared. For safety-critical applications, do not select this check box. If you want to select the **Remove protection against out-of-range index in generated code** check box, first check that your model inputs are in range. For example:

1. Clear the **Remove protection against out-of-range index in generated code** check box.
2. Set the **Diagnostic for out-of-range input** parameter to **Error**.
3. Simulate the model in normal mode.
4. If there are out-of-range errors, fix them to be in range and run the simulation again.
5. When the simulation no longer generates out-of-range input errors, select the **Remove protection against out-of-range index in generated code** check box.

**Note** When you select the **Remove protection against out-of-range index in generated code** check box and the input k or f is out-of-range, the behavior is undefined for generated code and simulations using accelerator mode.

Depending on your application, you can run the following Model Advisor checks to verify the usage of this check box:

- **By Product > Embedded Coder > Identify lookup table blocks that generate expensive out-of-range checking code**
- **By Product > Simulink Check > Modeling Standards > DO-178C/DO-331 Checks > Check usage of lookup table blocks**

For more information about the Model Advisor, see “Run Model Advisor Checks”.

**Programmatic Use**

**Block Parameter:** `RemoveProtectionInput`

**Type:** character vector

**Values:** `'off'` | `'on'`

**Default:** `'off'`

**Sample time — Specify sample time as a value other than -1**

- `-1` (default) | scalar | vector

Specify the sample time as a value other than -1. For more information, see “Specify Sample Time”.
Dependencies

This parameter is not visible unless it is explicitly set to a value other than -1. To learn more, see “Blocks for Which Sample Time Is Not Recommended”.

Programmatic Use
Block Parameter: SampleTime
Type: character vector
Values: scalar or vector
Default: '-1'

Data Types

Breakpoint — Breakpoint data type
Inherit: Same as input (default) | Inherit: Inherit from 'Breakpoint data' | double | single | int8 | uint8 | int16 | uint16 | int32 | uint32 | fixdt(1,16) | fixdt(1,16,0) | fixdt(1,16,2^0,0) | Enum: <class name> | <data type expression>

Specify the breakpoint data type. You can set it to:

- A rule that inherits a data type, for example, Inherit: Same as input
- The name of a built-in data type, for example, single
- The name of a data type class, for example, an enumerated data type class
- An expression that evaluates to a data type, for example, fixdt(1,16,0)

Click the Show data type assistant button to display the Data Type Assistant, which helps you set the data type attributes. For more information, see “Specify Data Types Using Data Type Assistant”.

Tip

- Specify a breakpoint data type different from the data type of input u for these cases:
  - Lower memory requirement for storing breakpoint data that uses a smaller type than the input signal u
  - Sharing of prescaled breakpoint data between two Prelookup blocks with different data types for input u
• Sharing of custom storage breakpoint data in the generated code for blocks with different data types for input \( u \)

• Enumerated data:
  
  • Breakpoints support unordered enumerated data. As a result, linear searches are also unordered, which offers flexibility but can impact performance. The search begins from the first element in the breakpoint.
  
  • If the **Begin index search using previous index result** check box is selected, you must use ordered monotonically increasing data. This ordering improves performance.
  
  • For enumerated data, **Extrapolation method** must be **Clip**.
  
  • Because the fraction is 1 or 0, select **Output selection > Index only**.

If you are using the index only output selection setting with the Interpolation Using Prelookup block, consider using the **Number of sub-table selection dimensions** parameter.

These are limitations for using enumerated data with this block:

• The block does not support out-of-range input for enumerated data. When specifying enumerated data, include the entire enumeration set in the breakpoint data set. For example, use the `enumeration` function.

• When breakpoints data source is set to **Inport port**, the enumeration data type must have 0 as the default value. For example, for this enumeration class, the default value of `GEAR1` must be 0.

```plaintext
classdef(Enumeration) Gears < Simulink.IntEnumType
  enumeration
    GEAR1(1),
    GEAR2(2),
    GEAR3(4),
    GEAR4(8),
    SPORTS(16),
    REVERSE(32),
    NEUTRAL(0)
  end
end
```
Dependencies

To enable this parameter, set the breakpoints data **Source** to **Dialog**.

**Note** When you set **Source** to **Input port**, the block inherits all breakpoint attributes (data type, minimum, and maximum) from the bp input port.

Programmatic Use

**Block Parameter:** BreakpointDataTypeStr  
**Type:** character vector  
**Values:** 'Inherit: Same as input' | 'Inherit: Inherit from 'Breakpoint data'' | 'double' | 'single' | 'int8' | 'uint8' | 'int16' | 'uint16' | 'int32' | 'uint32' | 'fixdt(1,16)' | 'fixdt(1,16,0)' | 'fixdt(1,16,2^0,0)' | '<data type expression>'  
**Default:** 'Inherit: Same as input'

**Index — Index data type**

uint32 (default) | int8 | uint8 | int16 | uint16 | int32 | uint32 | fixdt(1,16) | <data type expression>

Specify a data type that can index all elements in the breakpoint data set. You can:

- Select a built-in integer data type from the list.
- Specify an integer data type using a fixed-point representation.

Click the **Show data type assistant** button to display the **Data Type Assistant**, which helps you set the data type attributes. For more information, see “Specify Data Types Using Data Type Assistant”.

Programmatic Use

**Block Parameter:** IndexDataTypeStr  
**Type:** character vector  
**Values:** 'int8' | 'uint8' | 'int16' | 'uint16' | 'int32' | 'uint32' | 'fixdt(1,16)' | '<data type expression>'  
**Default:** 'uint32'

**Fraction — Fraction data type**

Inherit: Inherit via internal rule (default) | double | single | fixdt(1,16,0) | <data type expression>
Specify the data type of the interval fraction. You can:

- Select a built-in data type from the list.
- Specify data type inheritance through an internal rule.
- Specify a fixed-point data type using the [Slope Bias] or binary-point-only scaling representation.
  - If you use the [Slope Bias] representation, the scaling must be trivial — that is, the slope is 1 and the bias is 0.
  - If you use the binary-point-only representation, the fixed power-of-two exponent must be less than or equal to zero.

Click the **Show data type assistant** button to display the **Data Type Assistant**, which helps you set the data type attributes. For more information, see “Specify Data Types Using Data Type Assistant”.

**Dependencies**

This parameter displays only when you set **Output selection** on the **Main** tab to Index and fraction.

**Programmatic Use**

**Block Parameter:** FractionDataTypeStr  
**Type:** character vector  
**Values:** 'Inherit: Inherit via internal rule' | 'double' | 'single' | 'fixdt(1,16,0)' | '<data type expression>'  
**Default:** 'Inherit: Inherit via internal rule'

**Output — Output data type**

**Inherit:** auto (default) | **bus:** <object name>

To output a virtual bus, use the **Inherit:** auto setting. The resulting virtual bus contains two elements, the index and the fraction signals.

To output and specify a nonvirtual bus, use the **Bus:** <object name> template. Replace <object name> with the name of a bus object that contains the index and fraction signals.

- The bus object must contain two elements. The first element corresponds to the index signal and the second to the fraction signal.
The index and fraction bus element signals cannot be bus signals.

The data type and the complexity of the bus elements must meet the same constraints that apply to the index and fraction signals if you set **Output selection** to **Index** and **fraction**.

To create the bus object with the index and fraction bus elements, use MATLAB code similar to this, customizing the bus object name and the names and data types of the bus elements.

```matlab
% Bus object: kfBus
elems(1) = Simulink.BusElement;
elems(1).Name = 'Index';
elems(1).DataType = 'int8';

elems(2) = Simulink.BusElement;
elems(2).Name = 'Fraction';
elems(2).DataType = 'double';

kfBus = Simulink.Bus;
kfBus.Elements = elems;
clear elems;
```

Alternatively, you can use the Bus Editor to create or modify the bus object to use with the Prelookup block.

If you feed the bus output signal from this block to an Interpolation Using Prelookup block, select the **Require index and fraction as bus** check box in that block.

**Note** Use the Fixed-Point Tool data type override option to override bus objects with new bus objects that replace fixed-point data types with floating-point data types.

Overridden bus objects used with the Prelookup block can cause an error because the block does not accept floating-point data types for the first element in the bus.

If you encounter this issue, use the **Fix** button to redefine the original bus object and protect it from being overridden. For example, suppose you define the first element of the bus object to be an `int32`.

```matlab
myBus.Elements(1).DataType
int32
```

Clicking the **Fix** button redefines the first bus element:
myBus.Elements(1).DataType = 'fixdt(''int32'',''DataTypeOverride'',''Off'')'

Click the **Show data type assistant** button ![Show data type assistant](image) to display the **Data Type Assistant**, which helps you set the data type attributes. For more information, see “Specify Data Types Using Data Type Assistant”.

**Dependencies**

This parameter displays only when you set **Output selection** on the **Main** tab to **Index and fraction as bus**.

**Programmatic Use**

**Block Parameter:** `OutputBusDataTypeStr`

**Type:** character vector

**Values:**
- 'Inherit: auto'
- 'Bus: <object name>'
- '<data type expression>'

**Default:** 'Inherit: auto'

**Breakpoint Minimum — Minimum value breakpoint data can have**

[] (default) | scalar

Specify the minimum value that the breakpoint data can have. The default value is [] (unspecified).

**Dependencies**

To enable this parameter, set the breakpoints data **Source** to **Dialog** on the **Main** tab.

**Programmatic Use**

**Block Parameter:** `BreakpointMin`

**Type:** character vector

**Value:** scalar

**Default:** '[]'

**Breakpoint Maximum — Maximum value breakpoint data can have**

[] (default) | scalar

Specify the maximum value that the breakpoint data can have. The default value is [] (unspecified).
Dependencies

To enable this parameter, set the breakpoints data Source to Dialog on the Main tab.

Programmatic Use

Block Parameter: BreakpointMax
Type: character vector
Value: scalar
Default: '[]'

Lock output data type setting against changes by the fixed-point tools — Prevent fixed-point tools from overriding Output data type

off (default) | on

Select this parameter to prevent the fixed-point tools from overriding the Output data type you specify on the block. For more information, see “Use Lock Output Data Type Setting” (Fixed-Point Designer).

Programmatic Use

Block Parameter: LockScale
Type: character vector
Values: 'off' | 'on'
Default: 'off'

Integer rounding mode — Rounding mode for fixed-point operations

Floor (default) | Ceiling | Convergent | Nearest | Round | Simplest | Zero

Specify the rounding mode for fixed-point operations. For more information, see “Rounding” (Fixed-Point Designer).

Block parameters always round to the nearest representable value. To control the rounding of a block parameter, enter an expression using a MATLAB rounding function into the mask field.

Programmatic Use

Block Parameter: RndMeth
Type: character vector
Values: 'Ceiling' | 'Convergent' | 'Floor' | 'Nearest' | 'Round' | 'Simplest' | 'Zero'
Default: 'Floor'
Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>double</th>
<th>enumerated</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

More About

Enumerated Values in Prelookup

Simulate a Prelookup block with enumerated values.

Suppose that you have a Prelookup block with an enumerated class like this defined:

```matlab
classdef(Enumeration) Gears < Simulink.IntEnumType
    enumeration
        GEAR1(1),
        GEAR2(2),
        GEAR3(4),
        GEAR4(8),
        SPORTS(16),
        REVERSE(-1),
        NEUTRAL(0)
    end
end
```

Prelookup block has these settings:

- **Breakpoints data** value is `enumeration('Gears')`.
- **Output selection** is Index only.
- For an unordered search, set **Index search method** to Linear search and clear the **Begin index search using previous index result** check box.
**Extrapolation method** is Clip.

Interpolation using Prelookup block has these settings:

- **Number of dimensions** to 1.
- **Table data** value is [5 10 20 40 80 -5 0].
- **Interpolation method** is Flat.
- **Number of sub-table selection dimensions** is 1.

Simulation produces a vector [10 -5 80], which correspond to GEAR2, REVERSE, and SPORTS.

---

**Extended Capabilities**

**C/C++ Code Generation**
Generate C and C++ code using Simulink® Coder™.

**HDL Code Generation**
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

**HDL Architecture**
This block has a single, default HDL architecture.
### HDL Block Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConstrainedOutputPipeline</td>
<td>Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. For more details, see “ConstrainedOutputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td>InputPipeline</td>
<td>Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “InputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td>OutputPipeline</td>
<td>Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “OutputPipeline” (HDL Coder).</td>
</tr>
</tbody>
</table>

### Restrictions

#### Required Block Settings

- **Breakpoint data**: For **Source**, select **Dialog**.
- **Specification**: You can select either **Explicit values** or **Even spacing**.
- **Index search method**: Select **Evenly spaced points**.
- **Extrapolation method**: Select **Clip**.
- **Diagnostic for out-of-range input**: Select **Error**.
- **Use last breakpoint for input at or above upper limit**: Select this check box.
- **Breakpoint**: For **Data Type**, select **Inherit: Same as input**.
- **Integer rounding mode**: Select **Zero**, **Floor**, or **Simplest**.

#### Table Data Typing and Sizing

- It is good practice to structure your table such that the spacing between breakpoints is a power of two. If the breakpoint spacing does not meet this condition, HDL Coder issues a warning. When the breakpoint spacing is a power of two, you can replace division operations in the prelookup step with right-shift operations.
- All ports on the block require scalar values.
- The coder permits floating-point data for breakpoints.
**PLC Code Generation**
Generate Structured Text code using Simulink® PLC Coder™.

Simulink PLC Coder has limited support for lookup table blocks. The coder does not support:

- Number of dimensions greater than 2
- Cubic spline interpolation method
- Begin index search using a previous index mode
- Cubic spline extrapolation method

**Fixed-Point Conversion**
Design and simulate fixed-point systems using Fixed-Point Designer™.

**See Also**
Interpolation Using Prelookup | Simulink.Breakpoint | n-D Lookup Table

**Topics**
“About Lookup Table Blocks”
“Anatomy of a Lookup Table”
“Enter Breakpoints and Table Data”
“Guidelines for Choosing a Lookup Table”

**Introduced in R2006b**
Probe

Output signal attributes, including width, dimensionality, sample time, and complex signal flag

Library:  
Simulink / Signal Attributes
HDL Coder / Signal Attributes

Description

The Probe block outputs selected information about the signal on its input. The block can output the following attributes of the input signal: width, dimensionality, sample time, and a flag indicating whether the input is a complex-valued signal. The block has one input port. The number of output ports depends on the information that you select for probing, that is, signal dimensionality, sample time, and/or complex signal flag. Each probed value is output as a separate signal on a separate output port, with an independent data type control. During simulation, the block icon displays the probed data.

Ports

Input

Port_1 — Input signal
scalar | vector | matrix | N-D array

Input signal to probe, specified as a scalar, vector, matrix, or N-D array. The block accepts real or complex-valued signals of any built-in data type.

You can use an array of buses as an input signal to a Probe block. For details about defining and using an array of buses, see “Combine Buses into an Array of Buses”.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus
Output

Port_1 (W) — Signal width
scalar

Width, or number of elements, in the input signal, specified as a scalar. The width is also displayed on the block icon with the notation \( W \).

Dependencies

To enable this port, select Probe width.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

Port_2 (Ts) — Sample time
vector

Sample time of the input signal, as a two-element vector that specifies the period and offset of the sample time, respectively. The sample time is also displayed on the block icon with the notation \( Ts \). See “Specify Sample Time” for more information.

Dependencies

To enable this port, select Probe sample time.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

Port_3 (C) — Signal complexity
scalar

Indication of input signal complexity:

- When the input signal is complex, the block outputs 1.
- When the input signal is real-valued, the block outputs 0.

The indication of signal complexity is also displayed on the block icon with the notation \( C \).

Dependencies

To enable this port, select Detect complex signal.
Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

**Port_4 (D) — Signal dimensions**
scalar | vector

Dimensions of the input signal, output as a scalar or vector. The signal dimensions are also displayed on the block icon with the notation D:

**Dependencies**
To enable this port, select **Probe signal dimensions**.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

**Parameters**

**Main**

**Probe width — Output width of the input signal**
on (default) | off

Select to output the width, or number of elements, of the probed signal.

**Programmatic Use**

Block Parameter: ProbeWidth
Type: character vector
Values: 'off' | 'on'
Default: 'on'

**Probe sample time — Output sample time of input signal**
on (default) | off

Select to output the sample time of the probed signal. The output is a two-element vector that specifies the period and offset of the sample time, respectively. See “Specify Sample Time” for more information.

**Programmatic Use**

Block Parameter: ProbeSampleTime
Type: character vector
Values: 'off' | 'on'
Default: 'on'

Detect complex signal — Indicate the complexity of input signal
on (default) | off

Select to output 1 if the probed signal is complex; otherwise, 0.

Programmatic Use
Block Parameter: ProbeComplexSignal
Type: character vector
Values: 'off' | 'on'
Default: 'on'

Probe signal dimensions — Output dimensions of input signal
on (default) | off

Select to output the dimensions of the probed signal.

Programmatic Use
Block Parameter: ProbeSignalDimensions
Type: character vector
Values: 'off' | 'on'
Default: 'on'

Signal Attributes

Data type for width — Data type of signal width output
double (default) | single | int8 | uint8 | int16 | uint16 | int32 | uint32 | Same as input

Select the output data type for the signal width.

Programmatic Use
Block Parameter: ProbeWidthDataType
Type: character vector
Values: 'double' | 'single' | 'int8' | 'uint8' | 'int16' | 'uint16' | 'int32' | 'uint32' | 'Same as input'
Default: 'double'

Data type for sample time — Data type of sample time output
double (default) | single | int8 | uint8 | int16 | uint16 | int32 | uint32 | Same as input
Select the output data type for the sample time information.

**Programmatic Use**  
**Block Parameter:** ProbeSampleTimeDataType  
**Type:** character vector  
**Values:** 'double' | 'single' | 'int8' | 'uint8' | 'int16' | 'uint16' | 'int32' | 'uint32' | 'Same as input'  
**Default:** 'double'

**Data type for signal complexity — Data type of complexity output**  
`double` (default) | `single` | `int8` | `uint8` | `int16` | `uint16` | `int32` | `uint32` | `boolean` | `Same as input`

Select the output data type for the complexity information.

**Programmatic Use**  
**Block Parameter:** ProbeComplexityDataType  
**Type:** character vector  
**Values:** 'double' | 'single' | 'int8' | 'uint8' | 'int16' | 'uint16' | 'int32' | 'uint32' | 'Same as input'  
**Default:** 'double'

**Data type for signal dimensions — Data type for signal dimension output**  
`double` (default) | `single` | `int8` | `uint8` | `int16` | `uint16` | `int32` | `uint32` | `Same as input`

Select the output data type for the signal dimension output.

**Programmatic Use**  
**Block Parameter:** ProbeDimensionsDataType  
**Type:** character vector  
**Values:** 'double' | 'single' | 'int8' | 'uint8' | 'int16' | 'uint16' | 'int32' | 'uint32' | 'Same as input'  
**Default:** 'double'

## Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>bus</th>
<th>double</th>
<th>enumerated</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
<th>string</th>
</tr>
</thead>
</table>

1-1514
<table>
<thead>
<tr>
<th>Direct Feedthrough</th>
<th>yes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multidimensional Signals</td>
<td>yes</td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>yes</td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
</tr>
</tbody>
</table>

### Extended Capabilities

**C/C++ Code Generation**
Generate C and C++ code using Simulink® Coder™.

**HDL Code Generation**
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

**HDL Architecture**
This block has a single, default HDL architecture.

**HDL Block Properties**

| ConstrainedOutput Pipeline | Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. For more details, see “ConstrainedOutputPipeline” (HDL Coder). |
| **OutputPipeline** | Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “OutputPipeline” (HDL Coder). |

**Complex Data Support**

This block supports code generation for complex signals.

**Fixed-Point Conversion**

Design and simulate fixed-point systems using Fixed-Point Designer™.

**See Also**

Clock | Digital Clock | Weighted Sample Time Math

**Topics**

“Specify Sample Time”

**Introduced before R2006a**
Product

Multiply and divide scalars and nonscalars or multiply and invert matrices

Library:
- Simulink / Commonly Used Blocks
- Simulink / Math Operations
- HDL Coder / Commonly Used Blocks
- HDL Coder / HDL Floating Point Operations
- HDL Coder / Math Operations

Description

The Product block outputs the result of multiplying two inputs: two scalars, a scalar and a nonscalar, or two nonscalars that have the same dimensions. The default parameter values that specify this behavior are:

- **Multiplication**: Element-wise(\(.*\))
- **Number of inputs**: 2

This table shows the output of the Product block for example inputs using default block parameter values.

<table>
<thead>
<tr>
<th>Inputs and Behavior</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Scalar X Scalar</strong></td>
<td><img src="image" alt="Diagram" /></td>
</tr>
<tr>
<td>Output the product of the two inputs.</td>
<td><img src="image" alt="Result" /></td>
</tr>
</tbody>
</table>

1-1517
### Inputs and Behavior

<table>
<thead>
<tr>
<th></th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Scalar X Nonscalar</strong></td>
<td><img src="image1.png" alt="Diagram" /></td>
</tr>
<tr>
<td>Output a nonscalar having the same dimensions as the input nonscalar. Each element of the output nonscalar is the product of the input scalar and the corresponding element of the input nonscalar.</td>
<td></td>
</tr>
<tr>
<td><strong>Nonscalar X Nonscalar</strong></td>
<td><img src="image2.png" alt="Diagram" /></td>
</tr>
<tr>
<td>Output a nonscalar having the same dimensions as the inputs. Each element of the output is the product of corresponding elements of the inputs.</td>
<td></td>
</tr>
</tbody>
</table>

The Divide and Product of Elements blocks are variants of the Product block.

- For information on the Divide block, see Divide.
- For information on the Product of Elements block, see Product of Elements.

The Product block (or the Divide block or Product of Elements block, if appropriately configured) can:

- Numerically multiply and divide any number of scalar, vector, or matrix inputs
- Perform matrix multiplication and division on any number of matrix inputs

The Product block performs scalar or matrix multiplication, depending on the value of the **Multiplication** parameter. The block accepts one or more inputs, depending on the **Number of inputs** parameter. The **Number of inputs** parameter also specifies the operation to perform on each input.

The Product block can input any combination of scalars, vectors, and matrices for which the operation to perform has a mathematically defined result. The block performs the specified operations on the inputs, then outputs the result.

The Product block has two modes: **Element-wise mode**, which processes nonscalar inputs element by element, and **Matrix mode**, which processes nonscalar inputs as matrices.
Element-Wise Mode

When you set Multiplication to Element-wise(.*), the Product block is in Element-wise mode, in which it operates on the individual numeric elements of any nonscalar inputs. The MATLAB equivalent is the .* operator. In element-wise mode, the Product block can perform a variety of multiplication, division, and arithmetic inversion operations.

The value of the Number of inputs parameter controls both how many inputs exist and whether each is multiplied or divided to form the output. When the Product block is in element-wise mode and has only one input, it is functionally equivalent to a Product of Elements block. When the block has multiple inputs, any nonscalar inputs must have identical dimensions, and the block outputs a nonscalar with those dimensions. To calculate the output, the block first expands any scalar input to a nonscalar that has the same dimensions as the nonscalar inputs.

This table shows the output of the Product block for example inputs, using the indicated values for the Number of inputs parameter.

<table>
<thead>
<tr>
<th>Parameter Values</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of inputs: 2</td>
<td><img src="image1.png" alt="Image" /></td>
</tr>
<tr>
<td>Number of inputs: */</td>
<td><img src="image2.png" alt="Image" /></td>
</tr>
<tr>
<td>Number of inputs: /**/</td>
<td><img src="image3.png" alt="Image" /></td>
</tr>
</tbody>
</table>
### Parameter Values

<table>
<thead>
<tr>
<th></th>
<th>Examples</th>
</tr>
</thead>
</table>
| **Number of inputs:** | ![Diagram](image)

<table>
<thead>
<tr>
<th></th>
<th>Examples</th>
</tr>
</thead>
</table>
| **Number of inputs:** | ![Diagram](image)

### Matrix Mode

When the value of the **Multiplication** parameter is **Matrix(*)**, the Product block is in **Matrix mode**, in which it processes nonscalar inputs as matrices. The MATLAB equivalent is the * operator. In Matrix mode, the Product block can invert a single square matrix, or multiply and divide any number of matrices that have dimensions for which the result is mathematically defined.

The value of the **Number of inputs** parameter controls both how many inputs exist and whether each input matrix is multiplied or divided to form the output. The syntax of **Number of inputs** is the same as in element-wise mode. The difference between the modes is in the type of multiplication and division that occur.

### Expected Differences Between Simulation and Code Generation

For element-wise operations on complex floating-point inputs, simulation and code generation results might differ in near-overflow cases. Although complex numbers is selected and non-finite numbers is not selected on the **Code Generation > Interface** pane of the Configuration Parameters dialog box, the code generator does not emit special case code for intermediate overflows. This method improves the efficiency of embedded operations for the general case that does not include extreme values. If the inputs could include extreme values, you must manage these cases explicitly.

The generated code might not produce the exact same pattern of NaN and inf values as simulation when these values are mathematically meaningless. For example, if the
simulation output contains a \texttt{NaN}, output from the generated code also contains a \texttt{NaN}, but not necessarily in the same place.

**Ports**

**Input**

\textbf{Port\textsubscript{1} — First input to multiply or divide}

scalar | vector | matrix | N-D array

First input to multiply or divide, provided as a scalar, vector, matrix, or N-D array.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point

\textbf{Port\textsubscript{N} — Nth input to multiply or divide}

scalar | vector | matrix | N-D array

Nth input to multiply or divide, provided as a scalar, vector, matrix, or N-D array.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point

\textbf{X — Input signal to multiply}

scalar | vector | matrix | N-D array

Input signal to be multiplied with other inputs.

**Dependencies**

To enable one or more \textbf{X} ports, specify one or more * characters for the \texttt{Number of inputs} parameter.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point

\textbf{÷ — Input signal to divide or invert}

scalar | vector | matrix | N-D array

Input signal for division or inversion operations.
Dependencies

To enable one or more ÷ ports, specify one or more / characters for the **Number of inputs** parameter.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point

Output

**Port_1 — Output computed by multiplying, dividing, or inverting inputs**

scalar | vector | matrix | N-D array

Output computed by multiplying, dividing, or inverting inputs.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point

Parameters

Main

**Number of inputs — Control number of inputs and type of operation**

2 (default) | scalar | * or / for each input port

Control two properties of the block:

• The number of input ports on the block
• Whether each input is multiplied or divided into the output

When you specify:

• **1 or */

  The block has one input port. In element-wise mode, the block processes the input as described for the Product of Elements block. In matrix mode, if the parameter value is 1 or *, the block outputs the input value. If the value is /, the input must be a square matrix (including a scalar as a degenerate case) and the block outputs the matrix inverse. See “Element-Wise Mode” on page 1-1519 and “Matrix Mode” on page 1-1520 for more information.
• **Integer value > 1**
  
The block has the number of inputs given by the integer value. The inputs are multiplied together in element-wise mode or matrix mode, as specified by the **Multiplication** parameter. See “Element-Wise Mode” on page 1-1519 and “Matrix Mode” on page 1-1520 for more information.

• **Unquoted string of two or more * and / characters**
  
The block has the number of inputs given by the length of the character vector. Each input that corresponds to a * character is multiplied into the output. Each input that corresponds to a / character is divided into the output. The operations occur in element-wise mode or matrix mode, as specified by the **Multiplication** parameter. See “Element-Wise Mode” on page 1-1519 and “Matrix Mode” on page 1-1520 for more information.

**Programmatic Use**

**Block Parameter:** Inputs  
**Type:** character vector  
**Values:** ‘2’ | ‘**’ | ‘*/’ | ‘*/*’ | ...  
**Default:** ‘2’

**Multiplication — Element-wise (.* ) or Matrix (*) multiplication**  
Element-wise(.* ) (default) | Matrix(*)

Specify whether the block performs Element-wise(.* ) or Matrix(*) multiplication.

**Programmatic Use**

**Block Parameter:** Multiplication  
**Type:** character vector  
**Values:** 'Element-wise(.* )' | 'Matrix(*)'  
**Default:** 'Element-wise(.* )'

**Multiply over — All dimensions or specified dimension**  
All dimensions (default) | Specified dimension

Specify the dimension to multiply over as All dimensions, or Specified dimension. When you select Specified dimension, you can specify the **Dimension** as 1 or 2.

**Dependencies**

To enable this parameter, set **Number of inputs** to * and **Multiplication** to Element-wise (.* ).
Programmatic Use
Block Parameter: CollapseMode
Type: character vector
Values: 'All dimensions' | 'Specified dimension'
Default: 'All dimensions'

Dimension — Dimension to multiply over
1 (default) | 2 | ... | N

Specify the dimension to multiply over as an integer less than or equal to the number of dimensions of the input signal.

Dependencies
To enable this parameter, set:

- **Number of inputs** to *
- **Multiplication** to Element-wise (.*)
- **Multiply over** to Specified dimension

Programmatic Use
Block Parameter: CollapseDim
Type: character vector
Values: '1' | '2' | ...
Default: '1'

Sample time — Specify sample time as a value other than -1
-1 (default) | scalar | vector

Specify the sample time as a value other than -1. For more information, see “Specify Sample Time”.

Dependencies
This parameter is not visible unless it is explicitly set to a value other than -1. To learn more, see “Blocks for Which Sample Time Is Not Recommended”.

Programmatic Use
Block Parameter: SampleTime
Type: character vector
Values: scalar or vector
Default: '-1'
Signal Attributes

Require all inputs to have the same data type — Require that all inputs have the same data type
off (default) | on

Specify if input signals must all have the same data type. If you enable this parameter, then an error occurs during simulation if the input signal types are different.

Programmatic Use
Block Parameter: InputSameDT
Type: character vector
Values: 'off' | 'on'
Default: 'off'

Output minimum — Minimum output value for range checking
[ ] (default) | scalar

Lower value of the output range that Simulink checks.

Simulink uses the minimum to perform:

- Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters”) for some blocks.
- Simulation range checking (see “Signal Ranges” and “Enable Simulation Range Checking”).
- Automatic scaling of fixed-point data types.
- Optimization of the code that you generate from the model. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. For more information, see “Optimize using the specified minimum and maximum values” (Simulink Coder).

Note Output minimum does not saturate or clip the actual output signal. Use the Saturation block instead.

Programmatic Use
Block Parameter: OutMin
Type: character vector
Values: '[]' | scalar
Output maximum — Maximum output value for range checking

Upper value of the output range that Simulink checks.

Simulink uses the maximum value to perform:

• Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters”) for some blocks.
• Simulation range checking (see “Signal Ranges” and “Enable Simulation Range Checking”).
• Automatic scaling of fixed-point data types.
• Optimization of the code that you generate from the model. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. For more information, see “Optimize using the specified minimum and maximum values” (Simulink Coder).

Note Output maximum does not saturate or clip the actual output signal. Use the Saturation block instead.

Programmatic Use

Block Parameter: OutMax
Type: character vector
Values: '[ ]' | scalar
Default: '[ ]'

Output data type — Specify the output data type

Choose the data type for the output. The type can be inherited, specified directly, or expressed as a data type object such as Simulink.NumericType. For more information, see “Control Signal Data Types”.

When you select an inherited option, the block behaves as follows:
• **Inherit: Inherit via internal rule** — Simulink chooses a data type to balance numerical accuracy, performance, and generated code size, while taking into account the properties of the embedded target hardware. If you change the embedded target settings, the data type selected by the internal rule might change. For example, if the block multiplies an input of type `int8` by a gain of `int16` and ASIC/FPGA is specified as the targeted hardware type, the output data type is `sfix24`. If `Unspecified (assume 32-bit Generic)`, in other words, a generic 32-bit microprocessor, is specified as the target hardware, the output data type is `int32`. If none of the word lengths provided by the target microprocessor can accommodate the output range, Simulink software displays an error in the Diagnostic Viewer.

• **Inherit: Keep MSB**— Simulink chooses a data type that maintains the full range of the operation, then reduces the precision of the output to a size appropriate for the embedded target hardware.

  **Tip** For more efficient generated code, deselect the **Saturate on integer overflow** parameter.

  This rule never produces overflows.

• **Inherit: Match scaling**— Simulink chooses a data type whose scaling matches the scaling of the input types. If the full range of the type does not fit on the embedded target hardware, the range is reduced yielding a type appropriate for the embedded target hardware. This rule can produce overflows. This rule does not support multiplication between complex signals.

The **Inherit: Keep MSB** and **Inherit: Match scaling** rules do not support multiplication between complex signals or signals with non-zero bias. The rules support only multiplication and division (`'*'`, `'/'`, `'/'`) between two inputs, matrix multiplication of two inputs, and collapsing product of two elements of a vector.

It is not always possible for the software to optimize code efficiency and numerical accuracy at the same time. If the internal rule doesn’t meet your specific needs for numerical accuracy or performance, use one of the following options:

• Specify the output data type explicitly.
• Use the simple choice of **Inherit: Same as input**.
• Explicitly specify a default data type such as `fixdt(1,32,16)` and then use the Fixed-Point Tool to propose data types for your model. For more information, see `fxptdlg`.
• To specify your own inheritance rule, use Inherit: Inherit via back propagation and then use a Data Type Propagation block. Examples of how to use this block are available in the Signal Attributes library Data Type Propagation Examples block.

• Inherit: Inherit via back propagation — Use data type of the driving block.

• Inherit: Same as first input — Use data type of first input signal.

**Programmatic Use**

**Block Parameter:** OutDataTypeStr  
**Type:** character vector  
**Values:** 'Inherit: Inherit via internal rule' | 'Inherit: Keep MSB' | 'Inherit: Match scaling' | 'Inherit: Same as first input' | 'Inherit: Inherit via back propagation' | 'double' | 'single' | 'int8' | 'uint8' | 'int16' | 'uint16' | 'int32' | 'uint32' | 'int64' | 'uint64' | 'fixdt(1,16)' | 'fixdt(1,16,0)' | 'fixdt(1,16,2^0,0)' | '<data type expression>'  
**Default:** 'Inherit: Inherit via internal rule'

**Lock output data type setting against changes by the fixed-point tools — Prevent fixed-point tools from overriding Output data type**

**off** (default) | **on**

Select this parameter to prevent the fixed-point tools from overriding the **Output** data type you specify on the block. For more information, see “Use Lock Output Data Type Setting” (Fixed-Point Designer).

**Programmatic Use**

**Block Parameter:** LockScale  
**Type:** character vector  
**Values:** 'off' | 'on'  
**Default:** 'off'

**Integer rounding mode — Rounding mode for fixed-point operations**

**Floor** (default) | **Ceiling** | **Convergent** | **Nearest** | **Round** | **Simplest** | **Zero**

Select the rounding mode for fixed-point operations. You can select:

**Ceiling**

Rounds positive and negative numbers toward positive infinity. Equivalent to the MATLAB **ceil** function.
Convergent
Rounds number to the nearest representable value. If a tie occurs, rounds to the nearest even integer. Equivalent to the Fixed-Point Designer convergent function.

Floor
Rounds positive and negative numbers toward negative infinity. Equivalent to the MATLAB floor function.

Nearest
Rounds number to the nearest representable value. If a tie occurs, rounds toward positive infinity. Equivalent to the Fixed-Point Designer nearest function.

Round
Rounds number to the nearest representable value. If a tie occurs, rounds positive numbers toward positive infinity and rounds negative numbers toward negative infinity. Equivalent to the Fixed-Point Designer round function.

Simplest
Chooses between rounding toward floor and rounding toward zero to generate rounding code that is as efficient as possible.

Zero
Rounds number toward zero. Equivalent to the MATLAB fix function.

For more information, see “Rounding” (Fixed-Point Designer).

Block parameters always round to the nearest representable value. To control the rounding of a block parameter, enter an expression using a MATLAB rounding function into the mask field.

Programmatic Use

Block Parameter: RndMeth
Type: character vector
Values: 'Ceiling' | 'Convergent' | 'Floor' | 'Nearest' | 'Round' | 'Simplest' | 'Zero'
Default: 'Floor'

Saturate on integer overflow — Method of overflow action
off (default) | on

Specify whether overflows saturate or wrap.
<table>
<thead>
<tr>
<th>Action</th>
<th>Rationale</th>
<th>Impact on Overflows</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Select this check box (on).</td>
<td>Your model has possible overflow, and you want explicit saturation protection in the generated code.</td>
<td>Overflows saturate to either the minimum or maximum value that the data type can represent.</td>
<td>The maximum value that the int8 (signed, 8-bit integer) data type can represent is 127. Any block operation result greater than this maximum value causes overflow of the 8-bit integer. With the check box selected, the block output saturates at 127. Similarly, the block output saturates at a minimum output value of -128.</td>
</tr>
<tr>
<td>Do not select this check box (off).</td>
<td>You want to optimize efficiency of your generated code. You want to avoid overspecifying how a block handles out-of-range signals. For more information, see “Check for Signal Range Errors”.</td>
<td>Overflows wrap to the appropriate value that is representable by the data type.</td>
<td>The maximum value that the int8 (signed, 8-bit integer) data type can represent is 127. Any block operation result greater than this maximum value causes overflow of the 8-bit integer. With the check box cleared, the software interprets the overflow-causing value as int8, which can produce an unintended result. For example, a block result of 130 (binary 1000 0010) expressed as int8, is -126.</td>
</tr>
</tbody>
</table>

When you select this check box, saturation applies to every internal operation on the block, not just the output, or result. Usually, the code generation process can detect when overflow is not possible. In this case, the code generator does not produce saturation code.
Programmatic Use
Block Parameter: SaturateOnIntegerOverflow
Type: character vector
Values: 'off' | 'on'
Default: 'off'

Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>double</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.
HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.
HDL Architecture

The default Linear implementation generates a chain of N operations (multipliers) for N inputs.

HDL Block Properties

If you use the block in matrix multiplication mode, you can specify the DotProductStrategy. This setting determines whether you want to implement the matrix multiplication by using a tree of adders and multipliers, or use the Multiply-Accumulate block implementation. The default is Fully Parallel. For more information, see “DotProductStrategy” (HDL Coder).

See also “Design Considerations for Matrices and Vectors” (HDL Coder).

<table>
<thead>
<tr>
<th>General</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ConstrainedOutput Pipeline</strong></td>
</tr>
<tr>
<td><strong>DSPStyle</strong></td>
</tr>
<tr>
<td><strong>InputPipeline</strong></td>
</tr>
<tr>
<td><strong>OutputPipeline</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Native Floating Point</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>HandleDenormals</strong></td>
</tr>
</tbody>
</table>
Native Floating Point

<table>
<thead>
<tr>
<th>LatencyStrategy</th>
<th>Specify whether to map the blocks in your design to inherit, Max, Min, Zero, or Custom for the floating-point operator. The default is inherit. See also “LatencyStrategy” (HDL Coder).</th>
</tr>
</thead>
<tbody>
<tr>
<td>NFPCustomLatency</td>
<td>To specify a value, set LatencyStrategy to Custom. HDL Coder adds latency equal to the value that you specify for the NFPCustomLatency setting. See also “NFPCustomLatency” (HDL Coder).</td>
</tr>
<tr>
<td>MantissaMultiplyStrategy</td>
<td>Specify how to implement the mantissa multiplication operation during code generation. By using different settings, you can control the DSP usage on the target FPGA device. The default is inherit. See also “MantissaMultiplyStrategy” (HDL Coder).</td>
</tr>
</tbody>
</table>

Complex Data Support

The default (linear) implementation supports complex data.

Complex division is not supported. For block implementations of the Product block in divide mode or reciprocal mode, see “HDL Code Generation” on page 1-697 on the Divide block reference page.

Restrictions

HDL code generation does not support more than two inputs at the ports of the block when you use the block in matrix multiplication mode.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

Design and simulate fixed-point systems using Fixed-Point Designer™.

See Also

Divide | Dot Product | Product of Elements
Introduced before R2006a
**Product of Elements**

Copy or invert one scalar input, or collapse one nonscalar input

**Library:**
- Simulink / Math Operations
- HDL Coder / HDL Floating Point Operations
- HDL Coder / Math Operations

**Description**

The Product of Elements block inputs one scalar, vector, or matrix. You can use the block to:

- Copy a scalar input unchanged
- Invert a scalar input (divide 1 by it)
- Collapse a vector or matrix to a scalar by multiplying together all elements or taking successive inverses of the elements
- Collapse a matrix to a vector using one of these options:
  - Multiply together the elements of each row or column
  - Take successive inverses of the elements of each row or column

The Product of Elements block is functionally a Product block that has two preset parameter values:

- **Multiplication:** Element-wise(.*)
- **Number of inputs:** *

Setting nondefault values for either of those parameters can change a Product of Elements block to be functionally equivalent to a Product block or a Divide block.
Ports

Input

Port_1 — First input to multiply or divide
scalar | vector | matrix | N-D array

First input to multiply or divide, provided as a scalar, vector, matrix, or N-D array.
Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point

Port_N — Nth input to multiply or divide
scalar | vector | matrix | N-D array

Nth input to multiply or divide, provided as a scalar, vector, matrix, or N-D array.
Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point

X — Input signal to multiply
scalar | vector | matrix | N-D array

Input signal to be multiplied with other inputs.

Dependencies

To enable one or more X ports, specify one or more * characters for the Number of inputs parameter.
Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point

÷ — Input signal to divide or invert
scalar | vector | matrix | N-D array

Input signal for division or inversion operations.

Dependencies

To enable one or more ÷ ports, specify one or more / characters for the Number of inputs parameter.
Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point

Output

Port_1 — Output computed by multiplying, dividing, or inverting inputs

scalar | vector | matrix | N-D array

Output computed by multiplying, dividing, or inverting inputs.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point

Parameters

Main

Number of inputs — Control number of inputs and type of operation

* (default) | positive integer scalar | * or / for each input port

Control two properties of the block:

• The number of input ports on the block
• Whether each input is multiplied or divided into the output

When you specify:

• 1 or * or /

The block has one input port. In element-wise mode, the block processes the input as described for the Product of Elements block. In matrix mode, if the parameter value is 1 or *, the block outputs the input value. If the value is /, the input must be a square matrix (including a scalar as a degenerate case) and the block outputs the matrix inverse. See “Element-Wise Mode” on page 1-1519 and “Matrix Mode” on page 1-1520 for more information.

• Integer value > 1

The block has the number of inputs given by the integer value. The inputs are multiplied together in element-wise mode or matrix mode, as specified by the
**Multiplication** parameter. See “Element-Wise Mode” on page 1-1519 and “Matrix Mode” on page 1-1520 for more information.

- **Unquoted string of two or more * and / characters**

The block has the number of inputs given by the length of the character vector. Each input that corresponds to a * character is multiplied into the output. Each input that corresponds to a / character is divided into the output. The operations occur in element-wise mode or matrix mode, as specified by the **Multiplication** parameter. See “Element-Wise Mode” on page 1-1519 and “Matrix Mode” on page 1-1520 for more information.

**Programmatic Use**

**Block Parameter:** Inputs  
**Type:** character vector  
**Values:** 2 | * | ** | */ | */  
**Default:** *

**Multiplication — Element-wise (.* or Matrix (*) multiplication**

Element-wise(.*)(default) | Matrix(*)

Specify whether the block performs Element-wise(.* or Matrix(*) multiplication.

**Programmatic Use**

**Block Parameter:** Multiplication  
**Type:** character vector  
**Values:** 'Element-wise(.*)' | 'Matrix(*)'  
**Default:** 'Element-wise(.*)'

**Multiply over — All dimensions or specified dimension**

All dimensions (default) | Specified dimension

Specify the dimension to multiply over as All dimensions, or Specified dimension.

When you select All dimensions and select configuration parameter **Use algorithms optimized for row-major array layout**, Simulink enables row-major algorithms for simulation. To generate row-major code, set configuration parameter **Array layout** (Simulink Coder) to **Row-major** in addition to selecting **Use algorithms optimized for row-major array layout**. The column-major and row-major algorithms differ only in the multiplication order. In some cases, due to different operation order on the same data set, you might experience minor numeric differences in the outputs of column-major and row-major algorithms.
When you select Specified dimension, you can specify the Dimension as 1 or 2.

Dependencies

To enable this parameter, set Number of inputs to * and Multiplication to Element-wise (.*).

Programmatic Use
Block Parameter: CollapseMode
Type: character vector
Values: 'All dimensions' | 'Specified dimension'
Default: 'All dimensions'

Dimension — Dimension to multiply over

Specify the dimension to multiply over as an integer less than or equal to the number of dimensions of the input signal.

Dependencies

To enable this parameter, set:

- Number of inputs to *
- Multiplication to Element-wise (.*)
- Multiply over to Specified dimension

Programmatic Use
Block Parameter: CollapseDim
Type: character vector
Values: '1' | '2' | ...
Default: '1'

Sample time — Specify sample time as a value other than -1

Specify the sample time as a value other than -1. For more information, see “Specify Sample Time”.

Dependencies

This parameter is not visible unless it is explicitly set to a value other than -1. To learn more, see “Blocks for Which Sample Time Is Not Recommended”.
**Programmatic Use**
**Block Parameter:** SampleTime
**Type:** character vector
**Values:** scalar or vector
**Default:** '-1'

**Signal Attributes**

**Require all inputs to have the same data type — Require that all inputs have the same data type**
*off* (default) | *on*

Specify if input signals must all have the same data type. If you enable this parameter, then an error occurs during simulation if the input signal types are different.

**Programmatic Use**
**Block Parameter:** InputSameDT
**Type:** character vector
**Values:** 'off' | 'on'
**Default:** 'off'

**Output minimum — Minimum output value for range checking**

[] (default) | scalar

Lower value of the output range that Simulink checks.

Simulink uses the minimum to perform:

- Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters”) for some blocks.
- Simulation range checking (see “Signal Ranges” and “Enable Simulation Range Checking”).
- Automatic scaling of fixed-point data types.
- Optimization of the code that you generate from the model. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. For more information, see “Optimize using the specified minimum and maximum values” (Simulink Coder).

**Note** Output minimum does not saturate or clip the actual output signal. Use the Saturation block instead.
Programmatic Use
Block Parameter: OutMin
Type: character vector
Values: '[]' | scalar
Default: '[]'

Output maximum — Maximum output value for range checking
[] (default) | scalar

Upper value of the output range that Simulink checks.

Simulink uses the maximum value to perform:

- Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters”) for some blocks.
- Simulation range checking (see “Signal Ranges” and “Enable Simulation Range Checking”).
- Automatic scaling of fixed-point data types.
- Optimization of the code that you generate from the model. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. For more information, see “Optimize using the specified minimum and maximum values” (Simulink Coder).

Note Output maximum does not saturate or clip the actual output signal. Use the Saturation block instead.

Programmatic Use
Block Parameter: OutMax
Type: character vector
Values: '[]' | scalar
Default: '[]'

Output data type — Specify the output data type
Inherit: Inherit via internal rule (default) | Inherit: Inherit via back propagation | Inherit: Same as first input | double | single | int8 | uint8 | int16 | uint16 | int32 | uint32 | int64 | uint64 | fixdt(1,16) | fixdt(1,16,0) | fixdt(1,16,2^0,0) | <data type expression>
Choose the data type for the output. The type can be inherited, specified directly, or expressed as a data type object such as Simulink.NumericType. For more information, see “Control Signal Data Types”.

When you select an inherited option, the block behaves as follows:

- **Inherit: Inherit via internal rule** — Simulink chooses a data type to balance numerical accuracy, performance, and generated code size, while taking into account the properties of the embedded target hardware. If you change the embedded target settings, the data type selected by the internal rule might change. For example, if the block multiplies an input of type int8 by a gain of int16 and ASIC/FPGA is specified as the targeted hardware type, the output data type is sfix24. If Unspecified (assume 32-bit Generic), in other words, a generic 32-bit microprocessor, is specified as the target hardware, the output data type is int32. If none of the word lengths provided by the target microprocessor can accommodate the output range, Simulink software displays an error in the Diagnostic Viewer.

It is not always possible for the software to optimize code efficiency and numerical accuracy at the same time. If the internal rule doesn’t meet your specific needs for numerical accuracy or performance, use one of the following options:

- Specify the output data type explicitly.
- Use the simple choice of **Inherit: Same as input**.
- Explicitly specify a default data type such as fixdt(1,32,16) and then use the Fixed-Point Tool to propose data types for your model. For more information, see fxptdlg.
- To specify your own inheritance rule, use **Inherit: Inherit via back propagation** and then use a Data Type Propagation block. Examples of how to use this block are available in the Signal Attributes library Data Type Propagation Examples block.

- **Inherit: Inherit via back propagation** — Use data type of the driving block.
- **Inherit: Same as first input** — Use data type of first input signal.

**Programmatic Use**

**Block Parameter:** OutDataTypeStr  
**Type:** character vector  
**Values:** 'Inherit: Inherit via internal rule' | 'Inherit: Same as first input' | 'Inherit: Inherit via back propagation' | 'double' | 'single' | 'int8' | 'uint8' | 'int16' | 'uint16' | 'int32' | 'uint32' | 'int64' | 'uint64'
| 'fixdt(1,16)' | 'fixdt(1,16,0)' | 'fixdt(1,16,2^0,0)' | '<data type expression>'

**Default**: 'Inherit: Inherit via internal rule'

**Lock output data type setting against changes by the fixed-point tools** — Prevent fixed-point tools from overriding Output data type

**off** (default) | **on**

Select this parameter to prevent the fixed-point tools from overriding the Output data type you specify on the block. For more information, see “Use Lock Output Data Type Setting” (Fixed-Point Designer).

**Programmatic Use**

**Block Parameter**: LockScale

**Type**: character vector

**Values**: 'off' | 'on'

**Default**: 'off'

**Integer rounding mode** — Rounding mode for fixed-point operations

**Floor** (default) | Ceiling | Convergent | Nearest | Round | Simplest | Zero

Select the rounding mode for fixed-point operations. You can select:

**Ceiling**

Rounds positive and negative numbers toward positive infinity. Equivalent to the MATLAB ceil function.

**Convergent**

Rounds number to the nearest representable value. If a tie occurs, rounds to the nearest even integer. Equivalent to the Fixed-Point Designer convergent function.

**Floor**

Rounds positive and negative numbers toward negative infinity. Equivalent to the MATLAB floor function.

**Nearest**

Rounds number to the nearest representable value. If a tie occurs, rounds toward positive infinity. Equivalent to the Fixed-Point Designer nearest function.

**Round**

Rounds number to the nearest representable value. If a tie occurs, rounds positive numbers toward positive infinity and rounds negative numbers toward negative infinity. Equivalent to the Fixed-Point Designer round function.
Simplest

Chooses between rounding toward floor and rounding toward zero to generate rounding code that is as efficient as possible.

Zero

Rounds number toward zero. Equivalent to the MATLAB fix function.

For more information, see “Rounding” (Fixed-Point Designer).

Block parameters always round to the nearest representable value. To control the rounding of a block parameter, enter an expression using a MATLAB rounding function into the mask field.

**Programmatic Use**

**Block Parameter:** RndMeth  
**Type:** character vector  
**Values:** 'Ceiling' | 'Convergent' | 'Floor' | 'Nearest' | 'Round' | 'Simplest' | 'Zero'  
**Default:** 'Floor'

**Saturate on integer overflow — Method of overflow action**

off (default) | on

Specify whether overflows saturate or wrap.

<table>
<thead>
<tr>
<th>Action</th>
<th>Rationale</th>
<th>Impact on Overflows</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Select this check box (on).</td>
<td>Your model has possible overflow, and you want explicit saturation protection in the generated code.</td>
<td>Overflows saturate to either the minimum or maximum value that the data type can represent.</td>
<td>The maximum value that the int8 (signed, 8-bit integer) data type can represent is 127. Any block operation result greater than this maximum value causes overflow of the 8-bit integer. With the check box selected, the block output saturates at 127. Similarly, the block output saturates at a minimum output value of -128.</td>
</tr>
</tbody>
</table>
### Action
Do not select this check box (off).

### Rationale
You want to optimize efficiency of your generated code.
You want to avoid overspecifying how a block handles out-of-range signals. For more information, see “Check for Signal Range Errors”.

### Impact on Overflows
Overflows wrap to the appropriate value that is representable by the data type.

### Example
The maximum value that the int8 (signed, 8-bit integer) data type can represent is 127. Any block operation result greater than this maximum value causes overflow of the 8-bit integer. With the check box cleared, the software interprets the overflow-causing value as int8, which can produce an unintended result. For example, a block result of 130 (binary 1000 0010) expressed as int8, is -126.

When you select this check box, saturation applies to every internal operation on the block, not just the output, or result. Usually, the code generation process can detect when overflow is not possible. In this case, the code generator does not produce saturation code.

### Programmatic Use
**Block Parameter:** SaturateOnIntegerOverflow  
**Type:** character vector  
**Values:** 'off' | 'on'  
**Default:** 'off'

### Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>double</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Direct Feedthrough</strong></td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Multidimensional Signals</strong></td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1-1545
### Variable-Size Signals

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>yes</strong></td>
<td></td>
</tr>
</tbody>
</table>

### Zero-Crossing Detection

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>no</strong></td>
<td></td>
</tr>
</tbody>
</table>

## Algorithms

The Product of Elements block uses these algorithms to perform element-wise operations on inputs of floating-point, built-in integer, and fixed-point types.

<table>
<thead>
<tr>
<th>Input</th>
<th>Element-Wise Operation</th>
<th>Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Real scalar, u</td>
<td>Multiplication</td>
<td>$y = u$</td>
</tr>
<tr>
<td></td>
<td>Division</td>
<td>$y = 1/u$</td>
</tr>
<tr>
<td>Real vector or matrix with elements u1, u2, u3, ..., uN</td>
<td>Multiplication</td>
<td>$y = u_1u_2u_3\ldots u_N$</td>
</tr>
<tr>
<td></td>
<td>Division</td>
<td>$y = (\frac{1}{u_1}/u_2)/u_3\ldots/u_N$</td>
</tr>
<tr>
<td>Complex scalar, u</td>
<td>Multiplication</td>
<td>$y = u$</td>
</tr>
<tr>
<td></td>
<td>Division</td>
<td>$y = 1/u$</td>
</tr>
<tr>
<td>Complex vector or matrix with elements u1, u2, u3, ..., uN</td>
<td>Multiplication</td>
<td>$y = u_1u_2u_3\ldots u_N$</td>
</tr>
<tr>
<td></td>
<td>Division</td>
<td>$y = (\frac{1}{u_1}/u_2)/u_3\ldots/u_N$</td>
</tr>
</tbody>
</table>

If the specified dimension for element-wise multiplication or division is a row or column of a matrix, the algorithm applies to that row or column. Consider this model.

![Model diagram](image-url)
The top Product of Elements block collapses the matrix input to a scalar by taking successive inverses of the four elements:

\[ y = \left( \frac{\frac{1+i}{2}}{3} \right) \frac{1}{4-i} \frac{1}{5} \]

The bottom Product of Elements block collapses the matrix input to a vector by taking successive inverses along the second dimension:

- \[ y(1) = \left( \frac{1+i}{2} \right) \frac{1}{3} \]
- \[ y(2) = \left( \frac{1}{4-i} \right) \frac{1}{5} \]

### Extended Capabilities

#### C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

#### HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

#### HDL Architecture

HDL Coder supports Tree and Cascade architectures for Product or Product of Elements blocks that have a single vector input with multiple elements.

This block has multi-cycle implementations that introduce additional latency in the generated code. To see the added latency, view the generated model or validation model. See “Generated Model and Validation Model” (HDL Coder).

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Additional cycles of latency</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear (default)</td>
<td>0</td>
<td>Generates a linear chain of adders to compute the sum of products.</td>
</tr>
<tr>
<td>Architecture</td>
<td>Additional cycles of latency</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>-----------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>Tree</td>
<td>0</td>
<td>Generates a tree structure of adders to compute the sum of products.</td>
</tr>
<tr>
<td>Cascade</td>
<td>1, when block has a single vector input port.</td>
<td>This implementation optimizes latency * area and is faster than the Tree implementation. It computes partial products and cascades multipliers. See “Cascade Architecture Best Practices” (HDL Coder).</td>
</tr>
</tbody>
</table>

**Note** The Product of Element block does not support HDL code generation with double data types in the Native Floating Point mode.

**HDL Block Properties**

If you use the block in matrix multiplication mode, you can specify the **DotProductStrategy**. This setting determines whether you want to implement the matrix multiplication by using a tree of adders and multipliers, or use the Multiply-Accumulate block implementation. The default is **Fully Parallel**. For more information, see “DotProductStrategy” (HDL Coder).

See also “Design Considerations for Matrices and Vectors” (HDL Coder).

<table>
<thead>
<tr>
<th>General</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ConstrainedOutput Pipeline</td>
<td>Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. For more details, see “ConstrainedOutputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td>DSPStyle</td>
<td>Synthesis attributes for multiplier mapping. The default is <strong>none</strong>. See also “DSPStyle” (HDL Coder).</td>
</tr>
</tbody>
</table>
General

**InputPipeline**
Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “InputPipeline” (HDL Coder).

**OutputPipeline**
Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “OutputPipeline” (HDL Coder).

Native Floating Point

**HandleDenormals**
Specify whether you want HDL Coder to insert additional logic to handle denormal numbers in your design. Denormal numbers are numbers that have magnitudes less than the smallest floating-point number that can be represented without leading zeros in the mantissa. The default is inherit. See also “HandleDenormals” (HDL Coder).

**LatencyStrategy**
Specify whether to map the blocks in your design to inherit, Max, Min, Zero, or Custom for the floating-point operator. The default is inherit. See also “LatencyStrategy” (HDL Coder).

**NFPCustomLatency**
To specify a value, set LatencyStrategy to Custom. HDL Coder adds latency equal to the value that you specify for the NFPCustomLatency setting. See also “NFPCustomLatency” (HDL Coder).

**MantissaMultiplyStrategy**
Specify how to implement the mantissa multiplication operation during code generation. By using different settings, you can control the DSP usage on the target FPGA device. The default is inherit. See also “MantissaMultiplyStrategy” (HDL Coder).

Complex Data Support

The default (linear) implementation supports complex data.

Complex division is not supported. For block implementations of the Product block in divide mode or reciprocal mode, see “HDL Code Generation” on page 1-697 on the Divide block reference page.
PLC Code Generation
Generate Structured Text code using Simulink® PLCCoder™.

Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

See Also
Divide | Dot Product | Product

Introduced before R2006a
Pulse Generator

Generate square wave pulses at regular intervals

Library: Simulink / Sources

Description

The Pulse Generator block generates square wave pulses at regular intervals. The block waveform parameters, **Amplitude**, **Pulse Width**, **Period**, and **Phase delay**, determine the shape of the output waveform. The following diagram shows how each parameter affects the waveform.

The Pulse Generator block can emit scalar, vector, or matrix signals of any real data type. To emit a scalar signal, use scalars to specify the waveform parameters. To emit a vector or matrix signal, use vectors or matrices, respectively, to specify the waveform parameters. Each element of the waveform parameters affects the corresponding element of the output signal. For example, the first element of a vector amplitude parameter determines the amplitude of the first element of a vector output pulse. All the waveform parameters must have the same dimensions after scalar expansion. The data type of the output is the same as the data type of the **Amplitude** parameter.
The block output can be generated in time-based or sample-based modes, determined by the **Pulse type** parameter.

### Time-Based Mode

In time-based mode, Simulink computes the block output only at times when the output actually changes. This approach results in fewer computations for the block output over the simulation time period. Activate this mode by setting the **Pulse type** parameter to **Time based**.

The block does not support a time-based configuration that results in a constant output signal. Simulink returns an error if the parameters **Pulse Width** and **Period** satisfy either of these conditions:

\[
\text{Period} \times \frac{\text{Pulse Width}}{100} = 0
\]

\[
\text{Period} \times \frac{\text{Pulse Width}}{100} = \text{Period}
\]

Depending on the pulse waveform characteristics, the intervals between changes in the block output can vary. For this reason, a time-based Pulse Generator block has a variable sample time. The sample time color of such blocks is brown (see “View Sample Time Information” for more information).

Simulink cannot use a fixed-step solver to compute the output of a time-based pulse generator. If you specify a fixed-step solver for models that contain time-based pulse generators, Simulink computes a fixed sample time for the time-based pulse generators. Then the time-based pulse generators simulate as sample based.

If you use a fixed-step solver and the **Pulse type** is **Time based**, choose the step size such that the period, phase delay, and pulse width (in seconds) are integer multiples of the solver step size. For example, suppose that the period is 4 seconds, the pulse width is 75% (that is, 3 s), and the phase delay is 1 s. In this case, the computed sample time is 1 s. Therefore, choose a fixed-step size of 1 or a number that divides 1 exactly (for example, 0.25). To ensure this setting, select **auto** on the **Solver** pane of the Configuration Parameters dialog box.
**Sample-Based Mode**

In sample-based mode, the block computes its outputs at fixed intervals that you specify. Activate this mode by setting the **Pulse type** parameter to *Sample based*.

An important difference between the time-based and sample-based modes is that in time-based mode, the block output is based on simulation time, and in sample-based mode, the block output depends only on the simulation start, regardless of elapsed simulation time.

This block supports reset semantics in sample-based mode. For example, if a Pulse Generator block is in a resettable subsystem that hits a reset trigger, the block output resets to its initial condition.

**Ports**

**Output**

**Port 1 — Output signal**

*scalar | vector | matrix*

Generated square wave pulse signal specified by the parameters.

Data Types: `single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point`

**Parameters**

**Pulse type — Computational technique**

*Time based (default) | Sample based*

Computational technique to generate the type of square wave that this block generates, either time- or sample-based. Some parameters in the dialog box appear depending on whether you select time-based or sample-based.

**Programmatic Use**

**Block Parameter**: `PulseType`

**Type**: character vector

**Values**: 'Time based' | 'Sample based'
**Default:** 'Time based'

**Time (t) — Source of time variable**

*Use simulation time*(default)*|*Use external signal*

Specifies whether to use simulation time or an external signal as the source of values for the output pulse's time variable. If you specify an external source, the block displays an input port for connecting the source. The output pulse differs as follows:

- **Use simulation time:** The block generates an output pulse where the time variable equals the simulation time.
- **Use external signal:** The block generates an output pulse where the time variable equals the value from the input port, which can differ from the simulation time.

**Programmatic Use**

**Block Parameter:** TimeSource

*Type:* character vector

*Values:* 'Use simulation time' | 'Use external signal'

*Default:* 'Use simulation time'

**Amplitude — Signal amplitude**

1 (default) | scalar

Specify the amplitude of the signal.

**Programmatic Use**

**Block Parameter:** Amplitude

*Type:* character vector

*Value:* scalar

*Default:* '1'

**Period (secs) — Pulse period**

10 (default) | scalar

Pulse period specified in seconds if the pulse type is time-based. If the pulse type is sample-based, then the period is specified as the number of sample times.

**Programmatic Use**

**Block Parameter:** Period

*Type:* character vector

*Value:* scalar
Default: '10'

**Pulse width — Duty cycle**

5 (default) | scalar in the range [0,100]

Duty cycle specified as the percentage of the pulse period that the signal is on if time-based or as number of sample times if sample-based.

**Programmatic Use**

**Block Parameter:** PulseWidth

**Type:** character vector

**Value:** scalar

**Default:** '5'

**Phase delay (secs) — Delay before pulse**

0 (default) | scalar

Delay before the pulse is generated, specified in seconds, if the pulse type is time-based or as number of sample times if the pulse type is sample-based.

**Programmatic Use**

**Block Parameter:** PhaseDelay

**Type:** character vector

**Value:** scalar

**Default:** '0'

**Sample time — Length of sample time**

0 (default) | scalar | vector

Length of the sample time for this block in seconds. This parameter appears only if the block's pulse type is sample-based. See “Specify Sample Time”.

**Programmatic Use**

**Block Parameter:** SampleTime

**Type:** character vector

**Value:** scalar

**Default:** '0'

**Interpret vector parameters as 1-D — Treat vectors as 1-D**

on (default) | off

Select this check box to output a vector of length N if the **Constant value** parameter evaluates to an N-element row or column vector.
• When you select this check box, the block outputs a vector of length N if the Constant value parameter evaluates to an N-element row or column vector. For example, the block outputs a matrix of dimension 1-by-N or N-by-1.

• When you clear this check box, the block does not output a vector of length N if the Constant value parameter evaluates to an N-element row or column vector.

Programmatic Use
Block Parameter: VectorParams1D
Type: character vector
Values: 'on' | 'off'
Default: 'on'

Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>double</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

Cannot be used inside a triggered subsystem hierarchy.

These blocks do not reference absolute time when configured for sample-based operation. In time-based operation, they depend on absolute time.
PLC Code Generation
Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

See Also
Signal Generator | Waveform Generator

Introduced before R2006a
Push Button

Set value of parameter when button is pressed

Library: Simulink / Dashboard

Description

When you press the Push Button block during a simulation, the value of the connected block parameter changes to a specified value. Use the Push Button block with other Dashboard blocks to create an interactive dashboard to control your model.

Double-clicking the Push Button block does not open its dialog box during simulation and when the block is selected. To edit the block's parameters, you can use the Property Inspector, or you can right-click the block and select Block Parameters from the context menu.

Connecting Dashboard Blocks

Dashboard blocks do not use ports to make connections. To connect Dashboard blocks to variables and block parameters in your model, use connect mode. Connect mode facilitates the process of connecting Dashboard blocks in your model, especially when you want to connect multiple blocks at once. If you only want to connect a single Dashboard block, you can also use the Connection table in the block dialog box to make the connection.

Tip To facilitate understanding and debugging your model, you can modify Dashboard block connections in your model during normal and accelerator mode simulations.

Note Dashboard blocks cannot connect to variables until you update your model diagram. To connect Dashboard blocks to variables or modify variable values between opening your model and running a simulation, update your model diagram using Ctrl+D.

To enter connect mode, click the Connect button that appears above your unconnected Dashboard block when you pause on it.
In connect mode, when you select one or more signals or blocks, a list of parameters and signals available for connection appears. Select a signal or parameter from the list to connect the selected Dashboard block. To connect another Dashboard block, pause on the block and click the Connect button above it. Then, make a selection of signals and blocks in your model, and choose a signal or parameter to connect.
When you finish connecting the Dashboard blocks in your model, click the **Exit** button in the upper-right of the canvas to exit connect mode.

**Tip** You can hide the message shown on unconnected blocks using the `set_param` function with the `ShowInitialText` block parameter.

### Parameter Logging

Tunable parameters connected to Dashboard blocks are logged to the Simulation Data Inspector, where you can view parameter values along with logged signal data. You can access logged parameter data in the MATLAB workspace by exporting the parameter data from the Simulation Data Inspector UI or by using the `Simulink.sdi.exportRun` function. For more information about exporting data with the Simulation Data Inspector UI, see “Export Data from the Simulation Data Inspector”. The parameter data is stored in a `Simulink.SimulationData.Parameter` object, accessible as an element in the exported `Simulink.SimulationData.Dataset`. 
Limitations

- Dashboard blocks can only connect to real scalar signals.
- You cannot use the **Connection** table to connect a Dashboard block to a block that is commented out. When you connect a Dashboard block to a commented block using connect mode, the Dashboard block does not display the connected value until you uncomment the block.
- Dashboard blocks cannot connect to signals inside referenced models.
- When you simulate a model hierarchy, Dashboard blocks inside referenced models do not update.
- Dashboard blocks do not support rapid accelerator simulation.
- Some signals are not available during accelerator mode simulations. To view such signals on a Dashboard block, mark the signal for logging.
- Parameters specified by indexing a variable array do not appear in the **Connection** table. For example, a block parameter defined as `engine(1)` using the variable `engine` does not appear in the table.

To access the parameter in the **Connection** table, assign the indexed value to a scalar variable, such as `engine_1`. Then, use the scalar variable to define the block parameter.

- When you connect a Dashboard block to a variable or parameter during simulation, the data for that variable or parameter does not get logged to the Simulation Data Inspector. To log variable and parameter data to the Simulation Data Inspector, connect the Dashboard block to the variable or parameter prior to simulation.
- When you simulate a model in external mode with the **Default parameter behavior** set to **Inlined**, Dashboard blocks can appear to change parameter and variable values. However, the change does not propagate to the simulation. For example, the Gain block displays changes made to its **Gain** parameter using the Dashboard blocks, but the **Gain** value used in the simulation does not change.

Parameters

**Connection** — Select a variable or block parameter to connect
variable and parameter connection options

Select the variable or block parameter to control using the **Connection** table. Populate the **Connection** table by selecting one or more blocks in your model. Select the radio
button next to the variable or parameter you want to control, and click Apply. To facilitate understanding and debugging your model, you can connect Dashboard blocks to variables and parameters in your model during simulation.

**Note** To see workspace variables in the connection table, update the model diagram using Ctrl+D.

**Programmatic Use**

To programmatically connect a Dashboard block to a tunable parameter or a variable, use a Simulink.HMI.ParamSourceInfo object. The Simulink.HMI.ParamSourceInfo object contains four properties. Some of the properties apply to connecting Dashboard blocks to parameters, and some apply to connecting Dashboard blocks to variables. Not all fields have a value for a connection because a given Dashboard block connects to either a parameter or a variable.

**Block Parameter:** Binding  
**Type:** Simulink.HMI.ParamSourceInfo  
**Default:** []

**Button Text — Text displayed on button**  
*Button* (default) | character vector

The text displayed on the Push Button block in your model.

**Programmatic Use**

**Block Parameter:** ButtonText  
**Type:** string or character array  
**Default:** 'Button'

**On Value — Value assigned to parameter when button is pressed**  
1 (default) | scalar

The value assigned to the connected block parameter when the button is pressed.

**Programmatic Use**

**Block Parameter:** OnValue  
**Type:** string or character array  
**Default:** '1'

**Label — Block label position**  
*Hide* (default) | Bottom | Top
Position of the block label. When the block is connected to an element in the model, the label is the name of the connected element.

**Programmatic Use**

**Block Parameter:** LabelPosition

**Type:** character vector

**Values:** 'Hide' | 'Bottom' | 'Top'

**Default:** 'Hide'

---

**Block Characteristics**

<table>
<thead>
<tr>
<th>Data Types</th>
<th>double</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>no</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

**Extended Capabilities**

**C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

Dashboard blocks are ignored for code generation.

**See Also**

Rocker Switch | Slider Switch | Toggle Switch
Topics
“Tune and Visualize Your Model with Dashboard Blocks”
“Decide How to Visualize Simulation Data”

Introduced in R2015a
Quantizer

Discretize input at given interval

**Library:** Simulink / Discontinuities

**Description**

The Quantizer block discretizes the input signal using a quantization algorithm. The block uses a round-to-nearest method to map signal values to quantized values at the output that are defined by the **Quantization interval**. A smooth input signal can take on a stair-step shape after quantization.

This equation mathematically describes the round-to-nearest method:

\[
y = q \times \text{round}(u/q)
\]

where \( y \) is the quantized output, \( u \) is the input, and \( q \) is the **Quantization interval**.

**Ports**

**Input**

**Port_1 — Input signal to quantize**

scalar | vector

Input signal to the quantization algorithm.

Data Types: single | double

**Output**

**Port_1 — Quantized output signal**

scalar | vector

Output signal composed of discrete values, quantized with a round-to-nearest approach:
\[ y = q \times \text{round}(u/q) \]

where \( y \) is the output, \( u \) is the input, and \( q \) is the **Quantization interval**.

Data Types: single | double

### Parameters

**Quantization interval** — Interval that defines quantization levels for the output signal

\[ 0.5 \text{ (default)} | \text{scalar} | \text{vector} \]

Specify the quantization interval used in the algorithm. Quantization levels for the Quantizer block output are integer multiples of the **Quantization interval**.

**Programmatic Use**

- **Block Parameter**: QuantizationInterval
- **Type**: character vector
- **Value**: Any real or complex value
- **Default**: '0.5'

**Treat as gain when linearizing** — Specify the gain value

\[ \text{On} \text{ (default)} | \text{boolean} \]

The linearization commands in Simulink software treat this block as a gain in state space. Select this check box to cause the commands to treat the gain as 1. Clear the box to have the commands treat the gain as 0.

**Programmatic Use**

- **Block Parameter**: LinearizeAsGain
  - **Type**: character vector
  - **Value**: 'off' | 'on'
  - **Default**: 'on'

**Sample time** — Specify sample time as a value other than -1

\[ -1 \text{ (default)} | \text{scalar} | \text{vector} \]

Specify the sample time as a value other than -1. For more information, see “Specify Sample Time”.
Dependencies

This parameter is not visible unless it is explicitly set to a value other than -1. To learn more, see “Blocks for Which Sample Time Is Not Recommended”.

Programmatic Use
Block Parameter: SampleTime
Type: character vector
Values: scalar or vector
Default: '-1'

Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>double</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
</tr>
</tbody>
</table>

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also
Rate Limiter | Relay
Introduced before R2006a
Quarter Gauge

Display input value on quadrant scale
Library: Simulink / Dashboard

Description

The Quarter Gauge block displays the connected signal on a quadrant scale during simulation. You can use the Quarter Gauge block with other Dashboard blocks to build an interactive dashboard of controls and indicators for your model. The Quarter Gauge block provides an indication of the instantaneous value of the connected signal throughout simulation. You can modify the range of the Quarter Gauge block to fit your data. You can also customize the appearance of the Quarter Gauge block to provide more information about your signal. For example, you can color-code in-specification and out-of-specification ranges.

Connecting Dashboard Blocks

Dashboard blocks do not use ports to connect to signals. To connect Dashboard blocks to signals in your model, use connect mode. Connect mode facilitates the process of connecting Dashboard blocks in your model, especially when you want to connect multiple blocks at once. If you only want to connect a single Dashboard block, you can also use the Connection table in the block dialog box to make the connection.

Tip To facilitate understanding and debugging your model, you can modify Dashboard block connections in your model during normal and accelerator mode simulations.

To enter connect mode, click the Connect button that appears above your unconnected Dashboard block when you pause on it.
In connect mode, when you select one or more signals or blocks, a list of parameters and signals available for connection appears. Select a signal or parameter from the list to connect the selected Dashboard block. To connect another Dashboard block, pause on the block and click the **Connect** button above it. Then, make a selection of signals and blocks in your model, and choose a signal or parameter to connect.
When you finish connecting the Dashboard blocks in your model, click the **Exit** button in the upper-right of the canvas to exit connect mode.

**Tip** You can hide the message shown on unconnected blocks using the `set_param` function and the `ShowInitialText` block parameter.

**Limitations**

- Dashboard blocks can only connect to real scalar signals.
- You cannot use the **Connection** table to connect a Dashboard block to a block that is commented out. When you connect a Dashboard block to a commented block using connect mode, the Dashboard block does not display the connected value until the you uncomment the block.
- Dashboard blocks cannot connect to signals inside referenced models.
- When you simulate a model hierarchy, Dashboard blocks inside referenced models do not update.
- Dashboard blocks do not support rapid accelerator simulation.
• Some signals are not available during accelerator mode simulations. To view such signals on a Dashboard block, mark the signal for logging.

### Parameters

**Connection — Select a signal to connect and display**

Select the signal to connect using the Connection table. Populate the Connection table by selecting signals of interest in your model. Select the radio button next to the signal you want to display. Click Apply to connect the signal. To facilitate understanding and debugging your model, you can connect Dashboard blocks to signals in your model during simulation.

**Programmatic Use**

**Block Parameter:** Binding  
**Type:** Simulink.HMI.SignalSpecification  
**Default:** []

**Minimum — Minimum tick mark value**

0 (default) | scalar

Finite, real, double, scalar value specifying the minimum tick mark value for the arc. The minimum must be less than the value entered for the maximum.

**Programmatic Use**

To programmatically set the Minimum parameter, use a 1-by-3 vector containing values for the Minimum, Tick Interval, and Maximum parameters, in that order. To use the auto value for the Tick Interval, leave the Tick Interval position in the vector empty or specify -1.

**Block Parameter:** Limits  
**Type:** 1x3 vector  
**Default:** [0 -1 100]

**Maximum — Maximum tick mark value**

100 (default) | scalar

Finite, real, double, scalar value specifying the maximum tick mark value for the arc. The maximum must be greater than the value entered for the minimum.
**Programmatic Use**

To programmatically set the Maximum parameter, use a 1-by-3 vector containing values for the Minimum, Tick Interval, and Maximum parameters, in that order. To use the auto value for the Tick Interval, leave the Tick Interval position in the vector empty or specify -1.

**Block Parameter**: Limits  
**Type**: 1x3 vector  
**Default**: [0 -1 100]

**Tick Interval — Interval between major tick marks**  
*auto (default) | scalar*

Finite, real, positive, integer, scalar value specifying the interval of major tick marks on the arc. When set to auto, the block automatically adjusts the tick interval based on the minimum and maximum values.

**Programmatic Use**

To programmatically set the Tick Interval parameter, use a 1-by-3 vector containing values for the Minimum, Tick Interval, and Maximum parameters, in that order. To use the auto value for the Tick Interval, leave the Tick Interval position in the vector empty or specify -1.

**Block Parameter**: Limits  
**Type**: 1x3 vector  
**Default**: [0 -1 100]

**Scale Colors — Color indications on gauge arc**  
*colors for arc ranges*

Color specifications for ranges on the arc. Press the + button to add a color. For each color added, specify the minimum and maximum values of the range where you want to display that color.

**Programmatic Use**

To programmatically specify the Scale Colors parameter, use an array of structures with fields:

- **Min** — Minimum value for the color range on the scale.
- **Max** — Maximum value for the color range on the scale.
- **Color** — 1-by-3 vector of double values between 0 and 1 to specify the color for the range of the scale in the form \([r \ g \ b]\)
Include a structure in the array for each scale range for which you want to specify a color.

```matlab
range1.Min = 0;
range1.Max = 10;
range1.Color = [0 0 1];
range2.Min = 10;
range2.Max = 15;
range2.Color = [0 1 0];
scaleRanges = [range1 range2];
```

**Block Parameter:** ScaleColors  
**Type:** structure array  
**Default:** 0x1 struct array

**Label — Block label position**

Top (default) | Bottom | Hide

Position of the block label. When the block is connected to an element in the model, the label is the name of the connected element.

**Programmatic Use**

**Block Parameter:** LabelPosition  
**Type:** character vector  
**Values:** 'Top' | 'Bottom' | 'Hide'  
**Default:** 'Top'

**Block Characteristics**

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>double</th>
<th>enumerated</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.
Dashboard blocks are ignored for code generation.

See Also
Custom Gauge | Gauge | Half Gauge | Linear Gauge

Topics
“Tune and Visualize Your Model with Dashboard Blocks”
“Decide How to Visualize Simulation Data”

Introduced in R2015a
Queue, Entity Queue

Enqueue messages and entities
Library: Simulink / Messages & Events
SimEvents

Description

This block stores entities or messages in a queue, based on the order of arrival or priority. Each element at the head of the queue departs when the downstream block is ready to accept it. The Queue block and the Entity Queue block are the same blocks with different default values for the Overwrite the oldest element if queue is full check box.

You can specify the capacity of the queue, and the policy when the queue is full. The block supports three different message or queue sorting policies, first-in-first out (FIFO), last-in-first out (LIFO), and priority. The priority queue can be used only when the Overwrite the oldest element if queue is full check box is cleared.

Ports

Input

Port_1 — Input entity or message
scalar | vector | matrix

Input entity or message that carries scalar, bus, or vector data to enter the queue.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | enumerated | bus | fixed point

Output

Port_1 — Output entity or message
scalar | vector | matrix
Output port that allows entities or messages at the head of the queue to depart when a downstream block is ready to accept it.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | enumerated | bus | fixed point

**Port_d** — Number of entities that have departed the block
scalar

Number of entities that have departed the block.

**Dependencies**

To enable this port, select Overwrite the oldest element if queue is full check box, and click the Statistics tab and select the box labeled Number of entities departed, d.

Data Types: double

**Port_n** — Number of entities that have not yet departed the block
scalar

Number of entities that have not yet departed the block.

**Dependencies**

To enable this port, select Overwrite the oldest element if queue is full check box, and click the Statistics tab and select the box labeled Number of entities in block, n.

Data Types: double

**Port_w** — Average wait time for entities in the block
scalar

Average wait time for entities in the block.

**Dependencies**

To enable this port, select Overwrite the oldest element if queue is full check box, and click the Statistics tab and select the box labeled Average wait, w.

Data Types: double

**Port_l** — Average length of the entity queue
scalar

Port_l outputs the average length of the entity queue.
Dependencies

To enable this port, select **Overwrite the oldest element if queue is full** check box, and click the **Statistics** tab and select the box labeled **Average queue length, l**.

Data Types: double

**Port_ex — Number of entities extracted**

scalar

Number of entities that are pulled out of this block.

Dependencies

To enable this port, select **Overwrite the oldest element if queue is full** check box, and click the **Statistics** tab and select the box labeled **Number of entities extracted, ex**.

Data Types: double

Parameters

**Overwrite the oldest element if queue is full — Specify queue overwriting policy**

on (default for Simulink) | off (default for SimEvents)

Select this check box to choose between two queue overwriting policies.

- If you select the check box, an incoming message overwrites the oldest if the queue is full.

  This mode represents a simple message buffer that you can use to generate asynchronous communication between Simulink components and production code.

- If you clear the check box, the block does not accept new messages if the queue is full.

  In this mode, you can manipulate entity data using event actions and visualize statistics.

  To customize actions when entities or messages enter, exit, or are blocked, enter MATLAB code in the **Entry action**, **Exit action**, or **Blocked action** field of the **Event actions** tab. For an example, see “Manage Entities Using Event Actions” (SimEvents).
**Programmatic Use**

**Block Parameter:** QueueOverwriting

- **Type:** character vector
- **Values:** 'on' | 'off'
- **Default:** 'on' (for Simulink) and 'off' (for SimEvents)

**Capacity — Specify the capacity of the queue**

25 (default) | scalar

Specify the capacity of the queue.

**Programmatic Use**

**Block Parameter:** Capacity

- **Type:** character vector
- **Values:** '25' | scalar
- **Default:** '25'

**Queue type — Choose the queue type**

FIFO (default) | LIFO | Priority

Choose the queue type.

- **FIFO** — first-in-first-out
- **LIFO** — last-in-first-out
- **Priority** — store elements in order of priority, see “Sort by Priority” (SimEvents).
  Priority can be selected when you clear the **Overwrite the oldest element if queue is full** check box.

**Note** Priority queue does not support fixed point data type.

**Programmatic Use**

**Block Parameter:** QueueType

- **Type:** character vector
- **Values:** 'FIFO' | 'LIFO' | 'Priority'
- **Default:** 'FIFO'

**Multicast tag — Specify the tag when accepting entities broadcast via multicast sources**

A (default) | character vector
Specify the tag when accepting entities broadcast via multicast sources. The Entity Multicast block requires SimEvents license.

**Dependencies**

This parameter is visible when you clear the **Overwrite the oldest element if queue is full** check box, and set **Entity arrival source** to Multicast.

**Programmatic Use**

**Block Parameter**: MulticastTag  
**Type**: character vector  
**Values**: 'A' | character vector  
**Default**: 'A'

**Priority source — Specify which attribute of the entity determines its priority**

Specify which attribute of the entity determines its priority.

**Dependencies**

This parameter is visible when you clear the **Overwrite the oldest element if queue is full** check box, and set **Queue type** to Priority.

**Programmatic Use**

**Block Parameter**: PrioritySource  
**Type**: character vector  
**Values**: 'PriorityAttribute' | character vector  
**Default**: 'PriorityAttribute'

**Sorting direction — Choose the direction of sorting based on priority**

Choose the direction of sorting based on priority.

- **Ascending** — Elements with smaller priority values appear in front of the queue.  
- **Descending** — Elements with greater priority values appear in front of the queue.

**Dependencies**

This parameter is visible when you clear the **Overwrite the oldest element if queue is full** check box, and set **Queue type** to Priority.
Programmatic Use
Block Parameter: SortingDirection
Type: character vector
Values: 'Ascending' | 'Descending'
Default: 'Ascending'

Entity arrival source — Choose the source of arrival for the entities
Input port (default) | Multicast

Choose the source of arrival for the entities.

- Input port — Input port is source of messages or entities.
- Multicast — Entity Multicast block is source of entities. The Entity Multicast block requires SimEvents license.

Dependencies

This parameter is visible when you clear the Overwrite the oldest element if queue is full check box, and set Queue type to Priority.

Programmatic Use
Block Parameter: EntityArrivalSource
Type: character vector
Values: 'Input port' | 'Multicast'
Default: 'Input port'

Event action — Specify the behavior of the entity in certain events
Entry (default) | Exit | Blocked

Specify the behavior of the entity in certain events. Define the behavior in the Event action parameter. The Entry and the Exit actions are called just after the entity entry and just before entity exit. The Blocked action is called after an entity is blocked.

Note Event actions do not support fixed point data type.

Dependencies

Event actions are visible when you clear the Overwrite the oldest element if queue is full check box.
Programmatic Use
Block Parameter: EntryAction, ExitAction, BlockedAction
Type: character vector
Values: MATLAB code
Default: ''

Number of entities departed, \(d\) — Outputs the number of entities that have departed the block
off (default) | on

Number of entities that have departed the block.

Dependencies

Number of entities departed, \(d\) is visible when you clear the Overwrite the oldest element if queue is full check box.

Programmatic Use
Block Parameter: NumberEntitiesDeparted
Type: character vector
Values: 'on' | 'off'
Default: 'off'

Number of entities in block, \(n\) — Outputs the number of entities present in the block, that are yet to depart
off (default) | on

Number of entities present in the block, but which are yet to depart.

Dependencies

Number of entities in block, \(n\) is visible when you clear the Overwrite the oldest element if queue is full check box.

Programmatic Use
Block Parameter: NumberEntitiesInBlock
Type: character vector
Values: 'on' | 'off'
Default: 'off'

Average wait, \(w\) — Outputs the average wait time
off (default) | on
Sum of the wait times for entities departing the block divided by their total number. Wait time is the duration between the Entity Queue block entry and exit of an entity. For more information, see “Interpret SimEvents Models Using Statistical Analysis” (SimEvents).

Dependencies

Average wait, \( w \) is visible when you clear the **Overwrite the oldest element if queue is full** check box.

Programmatic Use

Block Parameter: AverageWait

Type: character vector

Values: 'on' | 'off'

Default: 'off'

Average queue length, \( l \) — Outputs the average length of the entity queue

off (default) | on

Accumulated time-weighted average queue size. The block computes this value by:

1. Multiplying the size of the queue by its duration to calculate time-weighted queue size
2. Summing up all time-weighted queue sizes and averaging them over total time

For more information, see “Interpret SimEvents Models Using Statistical Analysis” (SimEvents).

Dependencies

Average queue length, \( l \) is visible when you clear the **Overwrite the oldest element if queue is full** check box.

Programmatic Use

Block Parameter: AverageQueueLength

Type: character vector

Values: 'on' | 'off'

Default: 'off'

Number of entities extracted, \( ex \) — Number of entities extracted from this block

off (default) | on

Outputs the number of extracted entities which are pulled out from this block by the Entity Find block. The Entity Find block requires a SimEvents license. If the extracted
entity is the first entity in the queue, the next entity is set as the pending entity to leave the block. If an entity attribute defines the priority in a priority queue and it is modified by the Entity Find block, the queue is sorted again. When an entity is extracted, **Number of entities departed**, \( d \), **Number of entities in block**, \( n \), **Average wait**, \( w \), and **Average queue length**, \( l \) statistics are updated accordingly. For more information about finding and extracting entities, see “Find and Extract Entities in SimEvents Models” (SimEvents).

**Dependencies**

**Number of entities extracted**, \( ex \) is visible when you clear the **Overwrite the oldest element if queue is full** check box.

**Programmatic Use**

**Block Parameter**: NumEntitiesExtracted

**Type**: character vector

**Values**: 'on' | 'off'

**Default**: 'off'

### Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>bus</th>
<th>double</th>
<th>enumerated</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
<th>string</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Direct Feedthrough</strong></td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Multidimensional Signals</strong></td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Variable-Size Signals</strong></td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Zero-Crossing Detection</strong></td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

Code generation is not supported for event actions and statistics.

See Also
Receive | Send

Topics
“Behavior and Features of Queues” (SimEvents)

Introduced in R2016a
Radio Button

Select parameter value
Library: Simulink / Dashboard

Description

The Radio Button block lets you change the value of the connected parameter during simulation. You can specify a list of values and labels and then select the value for the parameter from that list. Use the Radio Button block with other Dashboard blocks to build an interactive dashboard of controls and indicators for your model.

Double-clicking the Radio Button block does not open its dialog box during simulation and when the block is selected. To edit the block's parameters, you can use the Property Inspector, or you can right-click the block and select Block Parameters from the context menu.

Connecting Dashboard Blocks

Dashboard blocks do not use ports to make connections. To connect Dashboard blocks to variables and block parameters in your model, use connect mode. Connect mode facilitates the process of connecting Dashboard blocks in your model, especially when you want to connect multiple blocks at once. If you only want to connect a single Dashboard block, you can also use the Connection table in the block dialog box to make the connection.

Tip To facilitate understanding and debugging your model, you can modify Dashboard block connections in your model during normal and accelerator mode simulations.

Note Dashboard blocks cannot connect to variables until you update your model diagram. To connect Dashboard blocks to variables or modify variable values between opening your model and running a simulation, update your model diagram using Ctrl+D.
To enter connect mode, click the **Connect** button that appears above your unconnected Dashboard block when you pause on it.

In connect mode, when you select one or more signals or blocks, a list of parameters and signals available for connection appears. Select a signal or parameter from the list to connect the selected Dashboard block. To connect another Dashboard block, pause on the block and click the **Connect** button above it. Then, make a selection of signals and blocks in your model, and choose a signal or parameter to connect.
When you finish connecting the Dashboard blocks in your model, click the **Exit** button in the upper-right of the canvas to exit connect mode.

**Tip** You can hide the message shown on unconnected blocks using the `set_param` function with the `ShowInitialText` block parameter.

### Parameter Logging

Tunable parameters connected to Dashboard blocks are logged to the Simulation Data Inspector, where you can view parameter values along with logged signal data. You can access logged parameter data in the MATLAB workspace by exporting the parameter data from the Simulation Data Inspector UI or by using the `Simulink.sdi.exportRun` function. For more information about exporting data with the Simulation Data Inspector UI, see “Export Data from the Simulation Data Inspector”. The parameter data is stored in a `Simulink.SimulationData.Parameter` object, accessible as an element in the exported `Simulink.SimulationData.Dataset`. 
Limitations

- Dashboard blocks can only connect to real scalar signals.
- You cannot use the Connection table to connect a Dashboard block to a block that is commented out. When you connect a Dashboard block to a commented block using connect mode, the Dashboard block does not display the connected value until you uncomment the block.
- Dashboard blocks cannot connect to signals inside referenced models.
- When you simulate a model hierarchy, Dashboard blocks inside referenced models do not update.
- Dashboard blocks do not support rapid accelerator simulation.
- Some signals are not available during accelerator mode simulations. To view such signals on a Dashboard block, mark the signal for logging.
- Parameters specified by indexing a variable array do not appear in the Connection table. For example, a block parameter defined as engine(1) using the variable engine does not appear in the table.
  
  To access the parameter in the Connection table, assign the indexed value to a scalar variable, such as engine_1. Then, use the scalar variable to define the block parameter.
- When you connect a Dashboard block to a variable or parameter during simulation, the data for that variable or parameter does not get logged to the Simulation Data Inspector. To log variable and parameter data to the Simulation Data Inspector, connect the Dashboard block to the variable or parameter prior to simulation.
- When you simulate a model in external mode with the Default parameter behavior set to Inlined, Dashboard blocks can appear to change parameter and variable values. However, the change does not propagate to the simulation. For example, the Gain block displays changes made to its Gain parameter using the Dashboard blocks, but the Gain value used in the simulation does not change.

Parameters

Connection — Select a variable or block parameter to connect
variable and parameter connection options

Select the variable or block parameter to control using the Connection table. Populate the Connection table by selecting one or more blocks in your model. Select the radio
button next to the variable or parameter you want to control, and click Apply. To facilitate understanding and debugging your model, you can connect Dashboard blocks to variables and parameters in your model during simulation.

**Note** To see workspace variables in the connection table, update the model diagram using Ctrl+D.

**Programmatic Use**

To programmatically connect a Dashboard block to a tunable parameter or a variable, use a Simulink.HMI.ParamSourceInfo object. The Simulink.HMI.ParamSourceInfo object contains four properties. Some of the properties apply to connecting Dashboard blocks to parameters, and some apply to connecting Dashboard blocks to variables. Not all fields have a value for a connection because a given Dashboard block connects to either a parameter or a variable.

**Block Parameter:** Binding
**Type:** Simulink.HMI.ParamSourceInfo
**Default:** []

**States — Set of states to select for connected parameter**
scalar and character vector

Set of states from which to select a value for the connected parameter. Each State consists of a **State Value** and a **State Label**.

- **State Value** — Value assigned to the connected variable or parameter when you select the state with the corresponding **Label**.
- **State Label** — Label for each state. You can use the **Label** to display the value the connected parameter takes when the switch is positioned at the bottom, or you can enter a descriptive text label.

Click the + button to add additional States.

The default configuration for the block includes these States.
States

<table>
<thead>
<tr>
<th>State Value</th>
<th>State Label</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Label1</td>
</tr>
<tr>
<td>1</td>
<td>Label2</td>
</tr>
<tr>
<td>2</td>
<td>Label3</td>
</tr>
</tbody>
</table>

Programmatic Use

To programmatically configure the States for a block, use an array of structures containing the fields Value and Label. Include a structure in the array for each state you want to configure on the block.

```matlab
state1.Value = 1;
state1.Label = 'State 1';
state2.Value = 2;
state2.Label = 'State 2';
radioStates = [state1 state2];
```

**Block Parameter:** States  
**Type:** structure  
**Default:** 3x1 structure array

Enumerated Data Type — Specify state values and labels using an enumerated data type

off (default) | on

You can use an enumerated data type that pairs a numeric value with each enumeration to configure the state values and labels for the block. To specify the states for the block using an enumerated data type, first select the **Enumerated Data Type** option. Then, specify the name of the enumerated data type in the text box. The definition for the specified enumerated data type must be saved on the MATLAB path or in the base workspace.

Example: myEnumType

Programmatic Use

To programmatically specify the state labels and values for the block using an enumerated data type, specify 'on' for the **UseEnumeratedDataType** parameter and the name of the enumerated data type for the **EnumeratedDataType** parameter.  
**Block Parameter:** UseEnumeratedDataType
Type: string or character array
Values: 'on' | 'off'
Default: 'off'
Block Parameter: EnumeratedDataType
Type: string or character array
Default: ''

Group Name — Radio Button group name
RadioButtonGroup (default) | character vector

Name for the group of values displayed on the Radio Button block. Unlike the Block Name and Label, the Group Name always shows on the Radio Button block.

Example: Input Amplitude

Programmatic Use
Block Parameter: ButtonGroupName
Type: string or character array
Default: 'Group'

Label — Block label position
Top (default) | Bottom | Hide

Position of the block label. When the block is connected to an element in the model, the label is the name of the connected element.

Programmatic Use
Block Parameter: LabelPosition
Type: character vector
Values: 'Top' | 'Bottom' | 'Hide'
Default: 'Top'

Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>double</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>no</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-----------------------</td>
<td>----</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

## Extended Capabilities

### C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

Dashboard blocks are ignored for code generation.

### See Also
Check Box | Combo Box | Rotary Switch

### Topics
“Tune and Visualize Your Model with Dashboard Blocks”
“Decide How to Visualize Simulation Data”

**Introduced in R2017b**
Ramp

Generate constantly increasing or decreasing signal
Library: Simulink / Sources

Description

The Ramp block generates a signal that starts at a specified time and value and changes by a specified rate. The block's Slope, Start time, and Initial output parameters determine the characteristics of the output signal. All must have the same dimensions after scalar expansion.

Ports

Output

Port_1 — Output signal
scalar | vector | matrix

Generated output ramp signal characterized by the Slope, Start time, and Initial output parameters.

Data Types: double

Parameters

Slope — Slope of signal
1 (default) | scalar | vector | matrix

Specify the rate of change of the generated signal.
**Programmatic Use**

**Block Parameter:** `slope`

*Type:* character vector  
*Values:* scalar  
*Default:* `'1'`

**Start time — Time output begins**

0 (default) | scalar

Specify the time at which the block begins generating the signal.

**Programmatic Use**

**Block Parameter:** `start`

*Type:* character vector  
*Values:* scalar  
*Default:* `'0'`

**Initial output — Initial value of output signal**

0 (default) | scalar | vector | matrix

Specify the initial value of the output signal.

**Programmatic Use**

**Block Parameter:** `InitialOutput`

*Type:* character vector  
*Values:* scalar  
*Default:* `'0'`

**Interpret vector parameters as 1-D — Treat vectors as 1-D**

on (default) | off

Select this check box to output a vector of length N if the Constant value parameter evaluates to an N-element row or column vector.

- When you select this check box, the block outputs a vector of length N if the Constant value parameter evaluates to an N-element row or column vector. For example, the block outputs a matrix of dimension 1-by-N or N-by-1.
- When you clear this check box, the block does not output a vector of length N if the Constant value parameter evaluates to an N-element row or column vector.
**Type:** character vector
**Values:** 'on' | 'off'
**Default:** 'on'

## Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>double</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>no</td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
</tr>
</tbody>
</table>

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Not recommended for production code.

### See Also

Pulse Generator | Repeating Sequence

Introduced before R2006a
Random Number

Generate normally distributed random numbers

Library: Simulink / Sources

Description

The Random Number block generates normally distributed random numbers. To generate uniformly distributed random numbers, use the Uniform Random Number block.

You can generate a repeatable sequence using any Random Number block with the same nonnegative seed and parameters. The seed resets to the specified value each time a simulation starts. By default, the block produces a sequence that has a mean of 0 and a variance of 1. To generate a vector of random numbers with the same mean and variance, specify the Seed parameter as a vector.

Avoid integrating a random signal, because solvers must integrate relatively smooth signals. Instead, use the Band-Limited White Noise block.

The numeric parameters of this block must have the same dimensions after scalar expansion. If you select the Interpret vector parameters as 1-D check box, and the numeric parameters are row or column vectors after scalar expansion, the block outputs a 1-D signal. If you clear the Interpret vector parameters as 1-D check box, the block outputs a signal of the same dimensionality as the parameters.

Ports

Output

Port_1 — Output signal

scalar | vector | matrix
Output signal that is the generated random numbers falling within a normal Gaussian distribution. The output is repeatable for a given seed.

Data Types: double

## Parameters

**Mean — Mean of random numbers**

0 (default) | scalar

Specify the mean of the random numbers generated.

**Programmatic Use**

Block Parameter: Mean  
Type: character vector  
Values: scalar  
Default: '0'

**Variance — Variance of random numbers**

1 (default) | scalar

Specify the variance of the random numbers.

**Programmatic Use**

Block Parameter: Variance  
Type: character vector  
Values: scalar  
Default: '1'

**Seed — Starting seed**

0 (default) | positive integer

Specify the starting seed for the random number generator.

The output of number generated is repeatable for a given seed.

**Programmatic Use**

Block Parameter: Seed  
Type: character vector  
Values: scalar  
Default: '0'
Sample time — Time between intervals
0.1 (default) | integer

Specify the time interval between samples. The default is 0.1, which matches the default sample time of the Band-Limited White Noise block. See “Specify Sample Time” in the Simulink documentation for more information.

Programmatic Use
Block Parameter: SampleTime
Type: character vector
Values: scalar
Default: '0.1'

Interpret vector parameters as 1-D — Treat vectors as 1-D
on (default) | off

Select this check box to output a vector of length N if the Constant value parameter evaluates to an N-element row or column vector.

- When you select this check box, the block outputs a vector of length N if the Constant value parameter evaluates to an N-element row or column vector. For example, the block outputs a matrix of dimension 1-by-N or N-by-1.

- When you clear this check box, the block does not output a vector of length N if the Constant value parameter evaluates to an N-element row or column vector.

Programmatic Use
Block Parameter: VectorParams1D
Type: character vector
Values: 'on' | 'off'
Default: 'on'

Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>double</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>yes</td>
</tr>
</tbody>
</table>
Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also
Band-Limited White Noise | Uniform Random Number

Introduced before R2006a
Rate Limiter

Limit rate of change of signal

Library: Simulink / Discontinuities

Description

The Rate Limiter block limits the first derivative of the signal passing through it. The output changes no faster than the specified limit. The derivative is calculated using this equation:

\[ rate = \frac{u(i) - y(i - 1)}{t(i) - t(i - 1)} \]

where \( u(i) \) and \( t(i) \) are the current block input and time, and \( y(i-1) \) and \( t(i-1) \) are the output and time at the previous step. The output is determined by comparing \( rate \) to the **Rising slew rate** and **Falling slew rate** parameters:

- If \( rate \) is greater than the **Rising slew rate** parameter \( (R) \), the output is calculated as
  \[ y(i) = \Delta t \cdot R + y(i - 1). \]
- If \( rate \) is less than the **Falling slew rate** parameter \( (F) \), the output is calculated as
  \[ y(i) = \Delta t \cdot F + y(i - 1). \]
- If \( rate \) is between the bounds of \( R \) and \( F \), the change in output is equal to the change in input:
  \[ y(i) = u(i) \]

When the block is running in continuous mode (for example, **Sample time mode** is inherited and **Sample time** of the driving block is zero), the **Initial condition** is ignored. The block output at \( t = 0 \) is equal to the initial input:

\[ y(0) = u(0) \]
When the block is running in discrete mode (for example, **Sample time mode** is inherited and **Sample time** of the driving block is nonzero), the **Initial condition** is preserved:

\[ y(-1) = I_c \]

where \( I_c \) is the initial condition. The block output at \( t = 0 \) is calculated as if \( \text{rate} \) is outside the bounds of \( R \) and \( F \). For \( t = 0 \), \( \text{rate} \) is calculated as follows:

\[ \text{rate} = \frac{u(0) - y(-1)}{\text{sample time}} \]

**Limitations**

- You cannot use a Rate Limiter block inside a Triggered Subsystem. Use the Rate Limiter Dynamic block instead.

**Ports**

**Input**

**Port_1 — Input signal**

scalar

The input signal to the rate limiter algorithm.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

**Output**

**Port_1 — Output signal**

scalar

Output signal from the rate limiter algorithm.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point
Parameters

**Rising slew rate — Limit of derivative for increasing input**
1 (default) | real number

Specify the limit of the rising rate of the input signal. This parameter is tunable for fixed-point inputs.

**Programmatic Use**
*Block Parameter:* RisingSlewLimit  
*Type:* character vector  
*Values:* real number  
*Default:* '1'

**Falling slew rate — Limit of derivative for decreasing input**
-1 (default) | real number

Specify the lower limit on the falling rate of the input signal. This parameter is tunable for fixed-point inputs.

**Programmatic Use**
*Block Parameter:* FallingSlewLimit  
*Type:* character vector  
*Values:* real number  
*Default:* '-1'

**Sample time mode — Sample time mode**
inherited (default) | continuous

Specify the sample time mode, continuous or inherited from the driving block.

**Programmatic Use**
*Block Parameter:* SampleTimeMode  
*Type:* character vector  
*Values:* 'inherited' | 'continuous' |  
*Default:* 'inherited'

**Initial condition — Initial output**
0 (default) | scalar

Set the initial output of the simulation. Simulink does not allow you to set the initial condition of this block to inf or NaN.
Programmatic Use
Block Parameter: InitialCondition
Type: character vector
Values: scalar
Default: '0'

Treat as gain when linearizing — Specify the gain value
On (default) | Boolean

Select this check box to cause the commands to treat the gain as 1. The linearization commands in Simulink software treat this block as a gain in state space. Clear the box to have the commands treat the gain as 0.

Programmatic Use
Block Parameter: LinearizeAsGain
Type: character vector
Values: 'off' | 'on'
Default: 'on'

Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>double</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Extended Capabilities

**C/C++ Code Generation**
Generate C and C++ code using Simulink® Coder™.

Cannot be used inside a triggered subsystem hierarchy.

**PLC Code Generation**
Generate Structured Text code using Simulink® PLC Coder™.

**Fixed-Point Conversion**
Design and simulate fixed-point systems using Fixed-Point Designer™.

**See Also**
Rate Limiter Dynamic | Saturation

*Introduced before R2006a*
Rate Limiter Dynamic

Limit rate of change of signal
Library: Simulink / Discontinuities

Description
The Rate Limiter Dynamic block limits the rising and falling rates of the signal.

- The external signal `up` sets the upper limit on the rising rate of the signal.
- The external signal `lo` sets the lower limit on the falling rate of the signal.

Follow these guidelines when using the Rate Limiter Dynamic block:

- Ensure that the data types of `up` and `lo` are the same as the data type of the input signal `u`.

  When the lower limit uses a signed type and the input signal uses an unsigned type, the output signal keeps increasing regardless of the input and the limits.

- Use a fixed-step solver to simulate models that contain this block.

  Because the Rate Limiter Dynamic block supports only discrete sample time.

Ports

Input

`u` — Input signal
scalar

Input signal to the rate limiter algorithm.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point
**lo — Limit of derivative for decreasing input**

Scalar

Dynamic value providing the limit of the falling rate of the input signal. Make the signal data type of lo the same data type of the input signal u.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

**up — Limit of derivative for increasing input**

Scalar

Dynamic value providing the limit of the rising rate of the input signal. Make the signal data type of up the same data type of the input signal u.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

**Output**

**Y — Output signal**

Scalar

Output signal from the rate limiter algorithm.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point | enumerated | bus

**Block Characteristics**

<table>
<thead>
<tr>
<th><strong>Data Types</strong></th>
<th>double</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Direct Feedthrough</strong></td>
<td>no</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Multidimensional Signals</strong></td>
<td>no</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Variable-Size Signals</strong></td>
<td>no</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1-1607
Zero-Crossing Detection

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

The code generator does not explicitly group primitive blocks that constitute a nonatomic masked subsystem block in the generated code. This flexibility allows for more efficient code generation. In certain cases, you can achieve grouping by configuring the masked subsystem block to execute as an atomic unit by selecting the Treat as atomic unit option.

PLC Code Generation
Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

See Also
Rate Limiter

Introduced before R2006a
Rate Transition

Handle transfer of data between blocks operating at different rates
Library: Simulink / Signal Attributes
        HDL Coder / Signal Attributes

Description

The Rate Transition block transfers data from the output of a block operating at one rate to the input of a block operating at a different rate. Use the block parameters to trade data integrity and deterministic transfer for faster response or lower memory requirements. To learn about data integrity and deterministic data transfer, see “Data Transfer Problems” (Simulink Coder).

Transition Handling Options

<table>
<thead>
<tr>
<th>Transition Handling Options</th>
<th>Block Parameter Settings</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Data integrity</td>
<td>Select:</td>
</tr>
<tr>
<td>• Deterministic data transfer</td>
<td>• Ensure data integrity during data transfer</td>
</tr>
<tr>
<td>• Maximum latency</td>
<td>• Ensure deterministic data transfer</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>• Data integrity</td>
<td>Select:</td>
</tr>
<tr>
<td>• Nondeterministic data transfer</td>
<td>• Ensure data integrity during data transfer</td>
</tr>
<tr>
<td>• Minimum latency</td>
<td></td>
</tr>
<tr>
<td>• Higher memory requirements</td>
<td>Clear:</td>
</tr>
<tr>
<td></td>
<td>• Ensure deterministic data transfer</td>
</tr>
</tbody>
</table>
### Transition Handling Options

- Potential loss of data integrity
- Nondeterministic data transfer
- Minimum latency
- Lower memory requirements

### Block Parameter Settings

<table>
<thead>
<tr>
<th>Clear:</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Ensure data integrity during data transfer</td>
</tr>
<tr>
<td>• Ensure deterministic data transfer</td>
</tr>
</tbody>
</table>

### Dependencies

The behavior of the Rate Transition block depends on:

- Sample times of the ports to which the block connects (see “Effects of Synchronous Sample Times” on page 1-1612 and “Effects of Asynchronous Sample Times” on page 1-1613)
- Priorities of the tasks for the source and destination sample times (see “Sample time properties” in the Simulink documentation)
- Whether the model specifies a fixed- or variable-step solver (see “Compare Solvers” in the Simulink documentation)

### Block Labels

When you update your diagram, a label appears on the Rate Transition block to indicate simulation behavior.

<table>
<thead>
<tr>
<th>Label</th>
<th>Block Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZOH</td>
<td>Acts as a zero-order hold</td>
</tr>
<tr>
<td>1/z</td>
<td>Acts as a unit delay</td>
</tr>
<tr>
<td>Buf</td>
<td>Copies input to output under semaphore control</td>
</tr>
<tr>
<td>Db_buf</td>
<td>Copies input to output using double buffers</td>
</tr>
<tr>
<td>3buf</td>
<td>Copies input to output using triple buffers</td>
</tr>
<tr>
<td>Copy</td>
<td>Unprotected copy of input to output</td>
</tr>
<tr>
<td>NoOp</td>
<td>Does nothing</td>
</tr>
<tr>
<td>Mixed</td>
<td>Expands to multiple blocks with different behaviors</td>
</tr>
<tr>
<td>Label</td>
<td>Block Behavior</td>
</tr>
<tr>
<td>-------</td>
<td>----------------</td>
</tr>
<tr>
<td>RT</td>
<td>Indicates data transfer between partitions when using the Schedule Editor. For more information on using the Schedule Editor, see “Using the Schedule Editor”.</td>
</tr>
</tbody>
</table>

The block behavior label shows the method that ensures safe transfer of data between tasks operating at different rates. You can use the sample-time colors feature (see “View Sample Time Information”) to display the relative rates that the block bridges. Consider, for example, the following model:

Sample-time colors and the block behavior label show:

- The Rate Transition block at the top of the diagram acts as a zero-order hold in a fast-to-slow transition.
- The Rate Transition block at the bottom of the diagram acts as a unit delay in a slow-to-fast transition.

For more information, see “Handle Rate Transitions” (Simulink Coder).
## Effects of Synchronous Sample Times

The following table summarizes how each label appears when the sample times of the input and output ports (\(\text{inTs}\) and \(\text{outTs}\)) are periodic, or synchronous.

<table>
<thead>
<tr>
<th>Block Settings</th>
<th>Conditions for Rate Transition Block</th>
<th>With Data Integrity and Determinism</th>
<th>With Only Data Integrity</th>
<th>Without Data Integrity or Determinism</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rate Transition</td>
<td>Block Label</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(\text{inTs} = \text{outTs}) (Equal)</td>
<td>in(\text{TsOffset} &lt; \text{outTsOffset})</td>
<td>None (error)</td>
<td>Buf</td>
<td>Copy or NoOp (see note that follows the table)</td>
</tr>
<tr>
<td></td>
<td>in(\text{TsOffset} = \text{outTsOffset})</td>
<td>Copy or NoOp (see note that follows the table)</td>
<td>Copy or NoOp (see note that follows the table)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>in(\text{TsOffset} &gt; \text{outTsOffset})</td>
<td>None (error)</td>
<td>Db_buf</td>
<td></td>
</tr>
<tr>
<td>(\text{inTs} &lt; \text{outTs}) (Fast to slow)</td>
<td>in(\text{Ts} = \frac{\text{outTs}}{N})</td>
<td>ZOH</td>
<td>Buf</td>
<td></td>
</tr>
<tr>
<td></td>
<td>in(\text{TsOffset}, \text{outTsOffset} = 0)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>in(\text{Ts} = \frac{\text{outTs}}{N})</td>
<td>None (error)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>in(\text{TsOffset} \leq \text{outTsOffset})</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>in(\text{Ts} = \frac{\text{outTs}}{N})</td>
<td>None (error)</td>
<td>Db_buf</td>
<td></td>
</tr>
<tr>
<td></td>
<td>in(\text{TsOffset} &gt; \text{outTsOffset})</td>
<td>None (error)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>in(\text{Ts} \neq \frac{\text{outTs}}{N})</td>
<td>None (error)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(\text{inTs} &gt; \text{outTs}) (Slow to fast)</td>
<td>in(\text{Ts} = \frac{\text{outTs}}{N})</td>
<td>1/z</td>
<td>Db_buf</td>
<td></td>
</tr>
<tr>
<td></td>
<td>in(\text{TsOffset}, \text{outTsOffset} = 0)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>in(\text{Ts} = \frac{\text{outTs}}{N})</td>
<td>None (error)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>in(\text{TsOffset} \leq \text{outTsOffset})</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>in(\text{Ts} = \frac{\text{outTs}}{N})</td>
<td>None (error)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>in(\text{TsOffset} &gt; \text{outTsOffset})</td>
<td>None (error)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>in(\text{Ts} \neq \frac{\text{outTs}}{N})</td>
<td>None (error)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
When you select the **Block reduction** parameter on the Configuration Parameters dialog box, Copy reduces to NoOp. No code generation occurs for a Rate Transition block with a NoOp label. To prevent a block from being reduced when block reduction is on, add a test point to the block output (see “Test Points” in the Simulink documentation).

## Effects of Asynchronous Sample Times

The following table summarizes how each label appears when the sample time of the input or output port (\(\text{inTs}\) or \(\text{outTs}\)) is not periodic, or asynchronous.

<table>
<thead>
<tr>
<th>Block Settings</th>
<th>Block Label</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{inTs} = \text{outTs})</td>
<td>Copy</td>
</tr>
<tr>
<td>(\text{inTs} \neq \text{outTs})</td>
<td>None (error)</td>
</tr>
</tbody>
</table>

**KEY**

- \(\text{inTs}\), \(\text{outTs}\): Sample times of input and output ports, respectively

## Ports

### Input

**Port_1 — Input signal**

- scalar
- vector
- matrix
- N-D array
Input signal to transition to a new sample rate, specified as a scalar, vector, matrix, or N-D array. To learn about the block parameters that enable you to trade data integrity and deterministic transfer for faster response or lower memory requirements, see “Transition Handling Options” on page 1-1609.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus

Output

Port_1  —  Output signal
scalar | vector | matrix | N-D array

Output signal is the input signal converted to the sample rate you specify. The default configuration ensures safe and deterministic data transfer. To learn about the block parameters that enable you to trade data integrity and deterministic transfer for faster response or lower memory requirements, see “Transition Handling Options” on page 1-1609.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus

Parameters

Ensure data integrity during data transfer  —  Ensure data integrity
on (default) | off

Selecting this check box results in generated code that ensures data integrity when the block transfers data. If you select this check box and the transfer is nondeterministic (see Ensure deterministic data transfer), depending on the priority of input rate and output rate, the generated code uses a proper algorithm using single or multiple buffers to protect data integrity during data transfer.

Otherwise, the Rate Transition block is either reduced or generates code using a copy operation to affect the data transfer. This unprotected mode consumes less memory. But the copy operation is also interruptible, which can lead to loss of data integrity during data transfers. Select this check box if you want the generated code to operate with maximum responsiveness (that is, nondeterministically) and data integrity. For more information, see “Rate Transition Block Options” (Simulink Coder).
**Programmatic Use**

**Block Parameter:** Integrity  
*Type:* character vector  
*Values:* `'off'` | `'on'`  
*Default:* `'on'`

**Ensure deterministic data transfer (maximum delay) — Ensure deterministic data transfer**  
*on* (default) | *off*  

Selecting this check box results in generated code that transfers data at the sample rate of the slower block, that is, deterministically. If you do not select this check box, data transfers occur when new data is available from the source block and the receiving block is ready to receive the data. You avoid transfer delays, thus ensuring that the system operates with maximum responsiveness. However, transfers can occur unpredictably, which is undesirable in some applications. For more information, see “Rate Transition Block Options” (Simulink Coder).

**Programmatic Use**

**Block Parameter:** Deterministic  
*Type:* character vector  
*Values:* `'off'` | `'on'`  
*Default:* `'on'`

**Initial conditions — Initial conditions**  
*0* (default) | scalar | vector | matrix | N-D array  

This parameter applies only to slow-to-fast transitions. It specifies the initial output of the Rate Transition block at the beginning of a transition when there is no output from the slow block connected to the input of the Rate Transition block. Simulink does not allow the initial output of this block to be `Inf` or `NaN`. The value you specify must be a scalar, or have the same dimensions as the input signal.

**Programmatic Use**

**Block Parameter:** InitialCondition  
*Type:* character vector  
*Values:* finite scalar  
*Default:* `'0'`

**Output port sample time options — Mode for specifying output port sample time**  
Specify (default)| Inherit | Multiple of input port sample time
Specifies a mode for setting the output port sample time. The options are:

- **Specify** — Allows you to use the **Output port sample time** parameter to specify the output rate to which the Rate Transition block converts its input rate.
- **Inherit** — Specifies that the Rate Transition block inherits an output rate from the block to which the output port is connected.
- **Multiple of input port sample time** — Allows you to use the **Sample time multiple (>0)** parameter to specify the Rate Transition block output rate as a multiple of its input rate.

If you select **Inherit** and all blocks connected to the output port also inherit sample time, the fastest sample time in the model applies.

**Programmatic Use**

**Block Parameter:** OutPortSampleTimeOpt  
**Type:** character vector  
**Values:** 'Specify' | 'Inherit' | 'Multiple of input port sample time'  
**Default:** 'Specify'

**Output port sample time — Output rate**

-1 (default) | scalar | vector

Enter a value that specifies the output rate to which the block converts its input rate. The default value (-1) specifies that the Rate Transition block inherits the output rate from the block to which the output port is connected. See “Specify Sample Time” in the Simulink documentation for information on how to specify the output rate.

**Dependencies**

To enable this parameter, set **Output port sample time options** to Specify.

**Programmatic Use**

**Block Parameter:** OutPortSampleTime  
**Type:** character vector  
**Values:** scalar | vector  
**Default:** '-1'

**Sample time multiple(>0) — Sample time multiple**

1 (default) | positive scalar

Enter a positive value that specifies the output rate as a multiple of the input port sample time. The default value (1) specifies that the output rate is the same as the input rate. A
value of 0.5 specifies that the output rate is half of the input rate. A value of 2 specifies that the output rate is twice the input rate.

**Dependencies**

To enable this parameter, set **Output port sample time options** to **Multiple of input port sample time**.

**Programmatic Use**

**Block Parameter:** OutPortSampleTimeMultiple  
**Type:** character vector  
**Values:** scalar  
**Default:** '1'

**Block Characteristics**

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>bus</th>
<th>double</th>
<th>enumerated</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Direct Feedthrough</th>
<th>yes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multidimensional Signals</td>
<td>yes</td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
</tr>
</tbody>
</table>

**Extended Capabilities**

**C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

- Generated code relies on memcpy or memset functions (string.h) under certain conditions
• Cannot use inside a triggered subsystem hierarchy

**HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

**Best Practices**

When the Rate Transition block is operating at a faster input rate and slower output rate, it is good practice to follow the Rate Transition block with a unit delay. Doing so prevents the code generator from inserting an extra bypass register in the HDL code.

See also “Multirate Model Requirements for HDL Code Generation” (HDL Coder).

**HDL Architecture**

This block has a single, default HDL architecture.

**HDL Block Properties**

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ConstrainedOutput Pipeline</strong></td>
<td>Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. For more details, see “ConstrainedOutputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td><strong>InputPipeline</strong></td>
<td>Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “InputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td><strong>OutputPipeline</strong></td>
<td>Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “OutputPipeline” (HDL Coder).</td>
</tr>
</tbody>
</table>

**Restrictions**

• Sample rate cannot be 0 or Inf for block input or output ports.
• **Ensure data integrity during data transfer** must be enabled.
- **Ensure deterministic data transfer (maximum delay) must be enabled.**

**Complex Data Support**

This block supports code generation for complex signals.

**Fixed-Point Conversion**

Design and simulate fixed-point systems using Fixed-Point Designer™.

**See Also**

Probes | Weighted Sample Time

**Topics**

“Specify Sample Time”
“View Sample Time Information”
“Handle Rate Transitions” (Simulink Coder)

**Introduced before R2006a**
Real-Imag to Complex

Convert real and/or imaginary inputs to complex signal

Library: Simulink / Math Operations
HDL Coder / Math Operations

Description

The Real-Imag to Complex block converts real and/or imaginary inputs to a complex-valued output signal.

The inputs can both be arrays (vectors or matrices) of equal dimensions, or one input can be an array and the other a scalar. If the block has an array input, the output is a complex array of the same dimensions. The elements of the real input map to the real parts of the corresponding complex output elements. The imaginary input similarly maps to the imaginary parts of the complex output signals. If one input is a scalar, it maps to the corresponding component (real or imaginary) of all the complex output signals.

Ports

Input

Re — Real part of complex output
scalar | vector | matrix

Real value to be converted to complex-valued output signal, specified as a scalar, vector, or matrix.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point | bus

Im — Imaginary part of complex output
scalar | vector | matrix
Imaginary value to be converted to complex-valued output signal, specified as a scalar, vector, or matrix.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point` | `bus`

**Port_1 — Real or imaginary part of complex output**

scalar | vector | matrix

Real or imaginary value to convert to complex output signal, specified as a finite scalar, vector, or matrix.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point` | `bus`

**Output**

**Port_1 — Complex signal**

scalar | vector | matrix

Complex signal, formed from real and imaginary values.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fixed point`

**Parameters**

**Input — Real, imaginary, or both**

Real (default) | Imag | Real and imag

Specify the type of input: a real input, an imaginary input, or both.

**Programmatic Use**

**Block Parameter:** Input

**Type:** character vector

**Values:** 'Real and imag' | 'Real' | 'Imag'

**Default:** 'Real and imag'

**Imag part — Imaginary part of complex output**

0 (default) | finite scalar, vector, or matrix
Specify the imaginary value to use when converting the input to a complex-valued output signal.

**Dependencies**

To enable this parameter, set **Input** to **Real**.

**Programmatic Use**

**Block Parameter:** ConstantPart  
**Type:** character vector  
**Values:** imaginary value  
**Default:** '0'

**Real part — Real part of complex output**

0 (default) | finite scalar, vector, or matrix

Specify the constant real value to use when converting the input to a complex-valued output signal.

**Dependencies**

To enable this parameter, set **Input** to **Imag**.

**Programmatic Use**

**Block Parameter:** ConstantPart  
**Type:** character vector  
**Values:** finite, real-valued scalar, vector, or matrix  
**Default:** '0'

**Sample time — Specify sample time as a value other than -1**

-1 (default) | scalar | vector

Specify the sample time as a value other than -1. For more information, see “Specify Sample Time”.

**Dependencies**

This parameter is not visible unless it is explicitly set to a value other than -1. To learn more, see “Blocks for Which Sample Time Is Not Recommended”.

**Programmatic Use**

**Block Parameter:** SampleTime  
**Type:** character vector  
**Values:** scalar or vector
Default: ' -1'

**Block Characteristics**

<table>
<thead>
<tr>
<th>Data Types</th>
<th>double</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Extended Capabilities**

**C/C++ Code Generation**
Generate C and C++ code using Simulink® Coder™.

**HDL Code Generation**
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

**HDL Architecture**
This block has a single, default HDL architecture.
HDL Block Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConstrainedOutput Pipeline</td>
<td>Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. For more details, see “ConstrainedOutputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td>InputPipeline</td>
<td>Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “InputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td>OutputPipeline</td>
<td>Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “OutputPipeline” (HDL Coder).</td>
</tr>
</tbody>
</table>

Complex Data Support

This block supports code generation for complex signals.

Fixed-Point Conversion

Design and simulate fixed-point systems using Fixed-Point Designer™.

See Also

Complex to Magnitude-Angle | Complex to Real-Imag | Magnitude-Angle to Complex

Topics

“Complex Signals”

Introduced before R2006a
Relational Operator

Perform specified relational operation on inputs

Library: Simulink / Commonly Used Blocks
         Simulink / Logic and Bit Operations
         HDL Coder / Commonly Used Blocks
         HDL Coder / HDL Floating Point Operations
         HDL Coder / Logic and Bit Operations

Description

The Relational Operator block performs the specified relational operation on the input. The value you choose for the Relational operator parameter determines whether the block accepts one or two input signals.

Two-Input Mode

By default, the Relational Operator block compares two inputs using the Relational operator parameter that you specify. The first input corresponds to the top input port and the second input to the bottom input port. (See “Port Location After Rotating or Flipping” for a description of the port order for various block orientations.)

You can specify one of the following operations in two-input mode:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>==</td>
<td>True if the first input is equal to the second input</td>
</tr>
<tr>
<td>~=</td>
<td>True if the first input is not equal to the second input</td>
</tr>
<tr>
<td>&lt;</td>
<td>True if the first input is less than the second input</td>
</tr>
<tr>
<td>&lt;=</td>
<td>True if the first input is less than or equal to the second input</td>
</tr>
<tr>
<td>&gt;=</td>
<td>True if the first input is greater than or equal to the second input</td>
</tr>
<tr>
<td>&gt;</td>
<td>True if the first input is greater than the second input</td>
</tr>
</tbody>
</table>

You can specify inputs as scalars, arrays, or a combination of a scalar and an array.
### For...

<table>
<thead>
<tr>
<th>For...</th>
<th>The output is...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scalar inputs</td>
<td>A scalar</td>
</tr>
<tr>
<td>Array inputs</td>
<td>An array of the same dimensions, where each element is the result of an element-by-element comparison of the input arrays</td>
</tr>
<tr>
<td>Mixed scalar and array inputs</td>
<td>An array, where each element is the result of a comparison between the scalar and the corresponding array element</td>
</tr>
</tbody>
</table>

The input with the smaller positive range is converted to the data type of the other input offline using round-to-nearest and saturation. This conversion occurs before the comparison.

You can specify the output data type using the **Output data type** parameter. The output equals 1 for true and 0 for false.

**Tip** Select an output data type that represents zero exactly. Data types that satisfy this condition include signed and unsigned integers and any floating-point data type.

### One-Input Mode

When you select one of the following operations for **Relational operator**, the block switches to one-input mode.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>isInf</td>
<td>True if the input is Inf</td>
</tr>
<tr>
<td>isNaN</td>
<td>True if the input is NaN</td>
</tr>
<tr>
<td>isFinite</td>
<td>True if the input is finite</td>
</tr>
</tbody>
</table>

For an input that is not floating point, the block produces the following output.

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Operation</th>
<th>Block Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fixed point</td>
<td>isInf</td>
<td>False</td>
</tr>
<tr>
<td>Boolean</td>
<td>isNaN</td>
<td>False</td>
</tr>
<tr>
<td>Built-in integer</td>
<td>isFinite</td>
<td>False</td>
</tr>
</tbody>
</table>
Rules for Data Type Propagation

The following rules apply for data type propagation when your block has one or more input ports with unspecified data types.

<table>
<thead>
<tr>
<th>When the block is in...</th>
<th>And...</th>
<th>The block uses...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Two-input mode</td>
<td>Both input ports have unspecified data types</td>
<td>double as the default data type for both inputs</td>
</tr>
<tr>
<td></td>
<td>One input port has an unspecified data type</td>
<td>The data type from the specified input port as the default data type of the other port</td>
</tr>
<tr>
<td>One-input mode</td>
<td>The input port has an unspecified data type</td>
<td>double as the default data type for the input</td>
</tr>
</tbody>
</table>

Ports

Input

**Port_1 — First input signal**
scalar | vector | matrix

First input signal, specified as a scalar, vector, or matrix.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated

**Port_2 — Second input signal**
scalar | vector | matrix

Second input signal, specified as a scalar, vector, or matrix.

Dependencies

To enable this port, set the Relational operator to ==, ~=, <, <=, >=, or >.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated
**Output**

**Port_1 — Output signal**

scalar | vector | matrix

Output signal, consisting of zeros and ones, with the same dimensions as the input. You control the output data type with the **Output data type** parameter.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated

**Parameters**

**Main**

**Relational operator — Relational operator**

\( \leq \) (default) | \(!=\) | \(==\) | \(<\) | \(\geq\) | \(>\) | isInf | isNaN | isFinite

Specify the operation for comparing two inputs or determining the signal type of one input.

- \(==\) — True if the first input is equal to the second input
- \(!=\) — True if the first input is not equal to the second input
- \(<\) — True if the first input is less than the second input
- \(\leq\) — True if the first input is less than or equal to the second input
- \(\geq\) — True if the first input is greater than or equal to the second input
- \(>\) — True if the first input is greater than the second input
- isInf — True if the input is Inf
- isNaN — True if the input is NaN
- isFinite — True if the input is finite

**Programmatic Use**

**Block Parameter:** Operator

**Type:** character vector

**Values:** '==' | '~=' | '<' | '<=' | '>=' | '>' | 'isInf' | 'isNaN' | 'isFinite'

1-1628
Enable zero-crossing detection — Enable zero-crossing detection
on (default) | off

Select to enable zero-crossing detection. For more information, see “Zero-Crossing Detection”.

Programmatic Use
Parameter: ZeroCross
Type: character vector, string
Values: 'on' | 'off'
Default: 'on'

Sample time — Specify sample time as a value other than -1
-1 (default) | scalar | vector

Specify the sample time as a value other than -1. For more information, see “Specify Sample Time”.

Dependencies
This parameter is not visible unless it is explicitly set to a value other than -1. To learn more, see “Blocks for Which Sample Time Is Not Recommended”.

Programmatic Use
Block Parameter: SampleTime
Type: character vector
Values: scalar or vector
Default: '-1'

Data Type

Require all inputs to have the same data type — Require all inputs to have same data type
off (default) | on

To require that all block inputs have the same data type, select this check box. When you clear this check box, the inputs can have different data types.
Dependencies

This check box is not available when you set Relational operator to isInf, isNaN, or isFinite because, in those modes, the block only has one input port.

Programmatic Use

**Block Parameter:** InputSameDT

*Type:* character vector

*Values:* 'off' | 'on'

*Default:* 'off'

**Output data type — Output data type**

boolean (default) | Inherit: Logical (see Configuration Parameters: Optimization) | fixdt(1,16) | <data type expression>

Specify the output data type. When you select:

- **boolean** — The block output has data type boolean.
- **Inherit: Logical (see Configuration Parameters: Optimization)** — The block uses the Implement logic signals as Boolean data configuration parameter to specify the output data type (see “Implement logic signals as Boolean data (vs. double)”).

**Note** This option supports models created before the boolean option was available. Use one of the other options, preferably boolean, for new models.

- **fixdt(1,16)** — The block output has the specified fixed-point data type fixdt(1,16).

**Tip** Click the Show data type assistant button to display the Data Type Assistant, which helps you set the data type attributes. For more information, see “Specify Data Types Using Data Type Assistant”.

- **<data type expression>** — The block output has the data type you specify as a data type expression, for example, Simulink.NumericType.

**Tip** To enter a built-in data type (double, single, int8, uint8, int16, uint16, int32, or uint32), enclose the expression in single quotes. For example, enter 'double' instead of double.
Programmatic Use

**Block Parameter:** OutDataTypeStr
**Type:** character vector
**Values:** 'Inherit: Logical (see Configuration Parameters: Optimization)' | 'boolean' | 'fixdt(1,16)' | '<data type expression>'
**Default:** 'boolean'

Integer rounding mode — Rounding mode for fixed-point operations

Specify the rounding mode for fixed-point operations. You can select:

- **Ceiling**
  Rounds positive and negative numbers toward positive infinity. Equivalent to the MATLAB `ceil` function.

- **Convergent**
  Rounds number to the nearest representable value. If a tie occurs, rounds to the nearest even integer. Equivalent to the Fixed-Point Designer `convergent` function.

- **Floor**
  Rounds positive and negative numbers toward negative infinity. Equivalent to the MATLAB `floor` function.

- **Nearest**
  Rounds number to the nearest representable value. If a tie occurs, rounds toward positive infinity. Equivalent to the Fixed-Point Designer `nearest` function.

- **Round**
  Rounds number to the nearest representable value. If a tie occurs, rounds positive numbers toward positive infinity and rounds negative numbers toward negative infinity. Equivalent to the Fixed-Point Designer `round` function.

- **Simplest**
  Chooses between rounding toward floor and rounding toward zero to generate rounding code that is as efficient as possible.

- **Zero**
  Rounds number toward zero. Equivalent to the MATLAB `fix` function.

For more information, see “Rounding” (Fixed-Point Designer).
**Programmatic Use**

**Block Parameter:** RndMeth  
**Type:** character vector  
**Values:** 'Ceiling' | 'Convergent' | 'Floor' | 'Nearest' | 'Round' | 'Simplest' | 'Zero'  
**Default:** 'Simplest'

### Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>double</th>
<th>enumerated</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Extended Capabilities

**C/C++ Code Generation**  
Generate C and C++ code using Simulink® Coder™.

**HDL Code Generation**  
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.
**HDL Architecture**

This block has a single, default HDL architecture.

**HDL Block Properties**

<table>
<thead>
<tr>
<th>General</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ConstrainedOutputPipeline</strong></td>
<td>Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. For more details, see “ConstrainedOutputPipeline” (HDL Coder).</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>InputPipeline</strong></td>
<td>Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “InputPipeline” (HDL Coder).</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>OutputPipeline</strong></td>
<td>Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “OutputPipeline” (HDL Coder).</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Native Floating Point**

<table>
<thead>
<tr>
<th>LatencyStrategy</th>
<th>Specify whether to map the blocks in your design to inherit, Max, Min, Zero, or Custom for the floating-point operator. The default is inherit. See also “LatencyStrategy” (HDL Coder).</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>NFPCustomLatency</strong></td>
<td>To specify a value, set LatencyStrategy to Custom. HDL Coderadds latency equal to the value that you specify for the NFPCustomLatency setting. See also “NFPCustomLatency” (HDL Coder).</td>
</tr>
</tbody>
</table>

**Complex Data Support**

The `!=` and `==` operators are supported for code generation.

**PLC Code Generation**

Generate Structured Text code using Simulink® PLC Coder™.
Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

See Also
Logical Operator

Introduced before R2006a
Relay

Switch output between two constants

**Library:**
- Simulink / Discontinuities
- HDL Coder / Discontinuities

**Description**

The output for the Relay block switches between two specified values. When the relay is on, it remains on until the input drops below the value of the **Switch off point** parameter. When the relay is off, it remains off until the input exceeds the value of the **Switch on point** parameter. The block accepts one input and generates one output.

**Note** When the initial input falls *between* the **Switch off point** and **Switch on point** values, the initial output is the value when the relay is off.

**Ports**

**Input**

**Port_1 — Input signal**

*scalar*

The input signal that switches the relay on or off.

*Data Types:* `single`, `double`, `int8`, `int16`, `int32`, `uint8`, `uint16`, `uint32`, `Boolean`, `fixed point`

**Output**

**Port_1 — Output signal**

*scalar*
The output signal switches between two values determined by the parameters **Output when on** and **Output when off**.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point

## Parameters

**Switch on point** — Input value which switches the relay on

'eps' (default) | scalar

When the input crosses this threshold, the relay switches on. The **Switch on point** value must be greater than or equal to the **Switch off point**. Specifying a **Switch on point** value greater than the **Switch off point** models hysteresis, whereas specifying equal values models a switch with a threshold at that value.

The **Switch on point** parameter is converted to the input data type offline using round-to-nearest and saturation methods.

**Programmatic Use**

**Block Parameter:** OnSwitchValue

**Type:** character vector

**Values:** scalar

**Default:** 'eps'

**Switch off point** — Input value which switches the relay off

'eps' (default) | scalar

When the input crosses this threshold the relay switches off. The value of **Switch off point** must be less than or equal to **Switch on point**. The **Switch off point** parameter is converted to the input data type offline using round-to-nearest and saturation.

**Programmatic Use**

**Block Parameter:** OffSwitchValue

**Type:** character vector

**Values:** scalar

**Default:** 'eps'

**Output when on** — Output value when the relay is on

1 (default) | scalar

The output value when the relay is on.
Programmatic Use

Block Parameter: OnOutputValue
Type: character vector
Values: scalar
Default: '1'

Output when off — Output value when the relay is off
0 (default) | scalar

The output value when the relay is off.

Programmatic Use

Block Parameter: OffOutputValue
Type: character vector
Values: scalar
Default: '0'

Input processing — Specify sample- or frame-based processing
Elements as channels (sample based) (default) | Columns as channels (frame based)

Specify whether the block performs sample- or frame-based processing:

• Columns as channels (frame based) — Treat each column of the input as a separate channel (frame-based processing).

Note Frame-based processing requires a DSP System Toolbox license.

For more information, see “Sample- and Frame-Based Concepts” (DSP System Toolbox).

• Elements as channels (sample based) — Treat each element of the input as a separate channel (sample-based processing).

Use Input processing to specify whether the block performs sample- or frame-based processing. The block accepts frame-based signals for the input u. All other input signals must be sample-based.

<table>
<thead>
<tr>
<th>Input Signal u</th>
<th>Input Processing Mode</th>
<th>Block Works?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sample based</td>
<td>Sample based</td>
<td>Yes</td>
</tr>
<tr>
<td>Frame based</td>
<td></td>
<td>No, produces an error</td>
</tr>
</tbody>
</table>

1-1637
<table>
<thead>
<tr>
<th>Input Signal u</th>
<th>Input Processing Mode</th>
<th>Block Works?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sample based</td>
<td>Frame based</td>
<td>Yes</td>
</tr>
<tr>
<td>Frame based</td>
<td>Frame based</td>
<td>Yes</td>
</tr>
</tbody>
</table>

For more information about these two processing modes, see “Sample- and Frame-Based Concepts” (DSP System Toolbox).

**Programmatic Use**

**Block Parameter:** InputProcessing  
*Type:* character vector  
*Values:* 'Columns as channels (frame based)' | 'Elements as channels (sample based)'

**Default:** 'Elements as channels (sample based)'

**Enable zero-crossing detection** — **Enable zero-crossing detection**

*on* (default) | Boolean

Select to enable zero-crossing detection. For more information, see “Zero-Crossing Detection” in the Simulink documentation.

**Programmatic Use**

**Block Parameter:** ZeroCross  
*Type:* character vector, string  
*Values:* 'off' | 'on'

**Default:** 'on'

**Sample time** — **Specify sample time as a value other than -1**

-1 (default) | scalar | vector

Specify the sample time as a value other than -1. For more information, see “Specify Sample Time”.

**Dependencies**

This parameter is not visible unless it is explicitly set to a value other than -1. To learn more, see “Blocks for Which Sample Time Is Not Recommended”.

**Programmatic Use**

**Block Parameter:** SampleTime  
*Type:* character vector  
*Values:* scalar or vector  
**Default:** '-1'
**Output minimum — Minimum output value for range checking**

[ ] (default) | scalar

Lower value of the output range that Simulink checks.

Simulink uses the minimum to perform:

- Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters”) for some blocks.
- Simulation range checking (see “Signal Ranges” and “Enable Simulation Range Checking”).
- Automatic scaling of fixed-point data types.
- Optimization of the code that you generate from the model. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. For more information, see “Optimize using the specified minimum and maximum values” (Simulink Coder).

**Note** Output minimum does not saturate or clip the actual output signal. Use the Saturation block instead.

---

**Programmatic Use**

**Block Parameter:** OutMin  
**Type:** character vector  
**Values:** `'[ ]'` | scalar  
**Default:** `'[ ]'`

---

**Output maximum — Maximum output value for range checking**

[ ] (default) | scalar

Upper value of the output range that Simulink checks.

Simulink uses the maximum value to perform:

- Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters”) for some blocks.
- Simulation range checking (see “Signal Ranges” and “Enable Simulation Range Checking”).
- Automatic scaling of fixed-point data types.
- Optimization of the code that you generate from the model. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. For more information, see “Optimize using the specified minimum and maximum values” (Simulink Coder).

**Note**: Output maximum does not saturate or clip the actual output signal. Use the Saturation block instead.

**Programmatic Use**

**Block Parameter**: OutMax

**Type**: character vector

**Values**: '[]' | scalar

**Default**: '[]'

**Output data type — Data type of output signal**

Inherit: All ports same datatype (default) | Inherit: Inherit via back propagation | double | single | int8 | int32 | uint32 | int64 | uint64 | fixdt(1,16,2^0,0) | <data type expression> | ...

Specify the output data type. You can set it to:

- A rule that inherits a data type, for example, Inherit: Inherit via back propagation
- The name of a built-in data type, for example, single
- The name of a data type object, for example, a Simulink.NumericType object
- An expression that evaluates to a data type, for example, fixdt(1,16,0)

Click the **Show data type assistant** button to display the **Data Type Assistant**, which helps you set the **Output data type** parameter.

See “Control Signal Data Types” in *Simulink User's Guide* for more information.

**Programmatic Use**

**Block Parameter**: OutDataTypeStr

**Type**: character vector

**Values**: 'Inherit: Same as input', 'Inherit: Inherit via back propagation', 'single', 'int8', 'uint8', 'int16', 'uint16', 'int32', 'uint32', etc.
'int64', 'uint64', fixdt(1,16,0), fixdt(1,16,2^0,0), fixdt(1,16,2^0,0).
'<data type expression>'
Default: 'Inherit: Same as input'

Lock data type settings against changes by the fixed-point tools — Prevent fixed-point tools from overriding data types
off (default) | on

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on this block. For more information, see “Lock the Output Data Type Setting” (Fixed-Point Designer).

Programmatic Use
Block Parameter: LockScale
Type: character vector
Values: 'off' | 'on'
Default: 'off'

Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>double</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.
HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture
This block has a single, default HDL architecture.

HDL Block Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConstrainedOutput Pipeline</td>
<td>Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. For more details, see “ConstrainedOutputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td>InputPipeline</td>
<td>Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “InputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td>OutputPipeline</td>
<td>Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “OutputPipeline” (HDL Coder).</td>
</tr>
</tbody>
</table>

PLC Code Generation
Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

See Also
Backlash | Saturation
Introduced before R2006a
Repeating Sequence

Generate arbitrarily shaped periodic signal
Library: Simulink / Sources

Description

The Repeating Sequence block outputs a periodic scalar signal having a waveform that you specify using the **Time values** and **Output values** parameters. The **Time values** parameter specifies a vector of output times. The **Output values** parameter specifies a vector of signal amplitudes at the corresponding output times. Together, the two parameters specify a sampling of the output waveform at points measured from the beginning of the interval over which the waveform repeats (the period of the signal).

By default, both parameters are `[0 2]`. These default settings specify a sawtooth waveform that repeats every 2 seconds from the start of the simulation and has a maximum amplitude of 2.

Ports

Output

**Port_1 — Periodic output signal**
scalar

Output signal specified by the **Time values** and **Output values** parameters to create a periodic scalar signal.

Data Types: double
Parameters

**Time values — Vector of output times**

Vector of strictly monotonically increasing time values. The period of the generated waveform is the difference of the last and first values of this parameter.

**Programmatic Use**

Block Parameter: rep_seq_t
Type: character vector
Values: vector
Default: [0 2]

**Output values — Vector of output values**

Vector of output values that specify the output waveform. Each element corresponds to the time value in the **Time values** parameter.

**Programmatic Use**

Block Parameter: rep_seq_y
Type: character vector
Values: vector
Default: [0 2]

Block Characteristics

<table>
<thead>
<tr>
<th><strong>Data Types</strong></th>
<th>double</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Direct Feedthrough</strong></td>
<td>no</td>
</tr>
<tr>
<td><strong>Multidimensional Signals</strong></td>
<td>no</td>
</tr>
<tr>
<td><strong>Variable-Size Signals</strong></td>
<td>no</td>
</tr>
<tr>
<td><strong>Zero-Crossing Detection</strong></td>
<td>no</td>
</tr>
</tbody>
</table>
Algorithms

The block sets the input period as the difference between the first and last value of the Time values parameter. The output at any time $t$ is the output at time $t = t - n \cdot \text{period}$, where $n$ is an integer. The sequence repeats at $t = n \cdot \text{period}$. The block uses linear interpolation to compute the value of the waveform between the output times that you specify.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Not recommended for production code.

Consider using Repeating Sequence Stair or Repeating Sequence Interpolated blocks for code generation.

See Also

Repeating Sequence Interpolated | Repeating Sequence Stair

Introduced before R2006a
Repeating Sequence Interpolated

Output discrete-time sequence and repeat, interpolating between data points

Library: Simulink / Sources

Description

The Repeating Sequence Interpolated block outputs a periodic discrete-time sequence based on the values in Vector of time values and Vector of output values parameters. Between data points, the block uses the method you specify for the Lookup Method parameter to determine the output.

Ports

Output

Port_1 — Periodic output signal
scalar

Output signal generated based on the values in the Vector of time values and Vector of output values parameters.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point
Parameters

Main

Vector of output values — Vector of output values
[3 1 4 2 1] (default) | vector

Vector of output values that specify the output waveform. Each element corresponds to the time value in the Vector of time values parameter.

Programmatic Use
Block Parameter: OutValues
Type: character vector
Values: vector
Default: [3 1 4 2 1].'

Vector of time values — Vector of time values
[0 0.1 0.5 0.6 1] (default) | vector

Specify the column vector containing time values. The time values must be strictly increasing, and the vector must have the same size as the vector of output values.

Programmatic Use
Block Parameter: TimeValues
Type: character vector
Values: vector
Default: [0 0.1 0.5 0.6 1].'

Lookup Method — Lookup method for output
Interpolation-Use End Values (default) | Use Input Nearest | Use Input Below | Use Input Above

Specify the lookup method to determine the output between data points.

Programmatic Use
Block Parameter: LookUpMeth
Type: character vector
Values: 'Interpolation-Use End Values' | 'Use Input Nearest' | Use Input Below | Use Input Above
Default: 'Interpolation-Use End Values'
Sample time — Time interval between samples
0.01 (default) | scalar

Specify the time interval between samples. To inherit the sample time, set this parameter to -1. See “Specify Sample Time” for more information.

Programmatic Use
Block Parameter: tsamp
Type: character vector
Values: scalar
Default: '0.01'

Signal Attributes

Output minimum — Minimum output value for range checking
[] (default) | scalar

Lower value of the output range that Simulink checks.

Simulink uses the minimum to perform:

- Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters”) for some blocks.
- Simulation range checking (see “Signal Ranges” and “Enable Simulation Range Checking”).
- Automatic scaling of fixed-point data types.
- Optimization of the code that you generate from the model. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. For more information, see “Optimize using the specified minimum and maximum values” (Simulink Coder).

Note Output minimum does not saturate or clip the actual output signal. Use the Saturation block instead.

Programmatic Use
Block Parameter: OutMin
Type: character vector
Values: '[ ]' | scalar
Default: '[]'

**Output maximum — Maximum output value for range checking**

[] (default) | scalar

Upper value of the output range that Simulink checks.

Simulink uses the maximum value to perform:

- Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters”) for some blocks.
- Simulation range checking (see “Signal Ranges” and “Enable Simulation Range Checking”).
- Automatic scaling of fixed-point data types.
- Optimization of the code that you generate from the model. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. For more information, see “Optimize using the specified minimum and maximum values” (Simulink Coder).

**Note** Output maximum does not saturate or clip the actual output signal. Use the Saturation block instead.

**Programmatic Use**

**Block Parameter:** OutMax
**Type:** character vector
**Values:** '[]' | scalar
**Default:** '[]'

**Output data type — Specify the output data type**

double (default) | Inherit: Inherit via back propagation | single | int8 | int32 | uint32 | int64 | uint64 | fixdt(1,16,2^0,0) | <data type expression> | ...

Choose the data type for the output. The type can be inherited, specified directly, or expressed as a data type object such as Simulink.NumericType.

**Programmatic Use**

**Block Parameter:** OutDataTypeStr
**Type:** character vector
Values: 'Inherit: Inherit via back propagation', 'single', 'int8', 'uint8', 'int16', 'uint16', 'int32', 'uint32', 'int64', 'uint64', fixdt(1,16,0), fixdt(1,16,2^0,0), fixdt(1,16,2^0,0). '<data type expression>'
Default: 'Double'

Lock output data type setting against changes by the fixed-point tools — Prevent fixed-point tools from overriding data types
off (default) | on

Select to lock the output data type setting of this block against changes by the Fixed-Point Tool and the Fixed-Point Advisor. For more information, see “Use Lock Output Data Type Setting” (Fixed-Point Designer).

Programmatic Use
Block Parameter: LockScale
Type: character vector
Values: 'off' | 'on'
Default: 'off'

Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>double</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

Cannot be used inside a triggered subsystem hierarchy.

PLC Code Generation
Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

See Also
Repeating Sequence | Repeating Sequence Stair

Introduced before R2006a
Repeating Sequence Stair

Output and repeat discrete time sequence

Library: Simulink / Sources

Description

The Repeating Sequence Stair block outputs and repeats a stair sequence that you specify with the Vector of output values parameter. For example, you can specify the vector as [3 1 2 4 1]'. A value in Vector of output values is output at each time interval, and then the sequence repeats.
Port_1 — Repeating discrete output signal
scalar
Output signal generated based on the values in the Vector of time values and Sample time parameters.
Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated
Parameters

Main

Vector of output values — Vector of output values
[3 1 4 2 1].' (default) | vector

Specify the vector containing values of the repeating stair sequence.

Programmatic Use
Block Parameter: OutValues
Type: character vector
Values: vector
Default: [3 1 4 2 1].'

Sample time — Time interval between samples
-1 (default) | scalar

Specify the time interval between samples. To inherit the sample time, set this parameter to -1. See “Specify Sample Time” for more information.

Programmatic Use
Block Parameter: tsamp
Type: character vector
Values: scalar
Default: '-1'

Signal Attributes

Output minimum — Minimum output value for range checking
[] (default) | scalar

Lower value of the output range that Simulink checks.

Simulink uses the minimum to perform:

- Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters”) for some blocks.
- Simulation range checking (see “Signal Ranges” and “Enable Simulation Range Checking”).
• Automatic scaling of fixed-point data types.
• Optimization of the code that you generate from the model. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. For more information, see “Optimize using the specified minimum and maximum values” (Simulink Coder).

**Note** Output minimum does not saturate or clip the actual output signal. Use the Saturation block instead.

**Programmatic Use**

**Block Parameter:** OutMin

**Type:** character vector

**Values:** '[]' | scalar

**Default:** '[]'

**Output maximum — Maximum output value for range checking**

[] (default) | scalar

Upper value of the output range that Simulink checks.

Simulink uses the maximum value to perform:

• Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters”) for some blocks.
• Simulation range checking (see “Signal Ranges” and “Enable Simulation Range Checking”).
• Automatic scaling of fixed-point data types.
• Optimization of the code that you generate from the model. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. For more information, see “Optimize using the specified minimum and maximum values” (Simulink Coder).

**Note** Output maximum does not saturate or clip the actual output signal. Use the Saturation block instead.
**Programmatic Use**

**Block Parameter:** OutMax

**Type:** character vector

**Values:** ' [ ]' | scalar

**Default:** ' [ ]'

### Output data type — Specify the output data type

double (default) | Inherit: Inherit via back propagation | single | int8 | int32 | uint32 | int64 | uint64 | fixdt(1,16,2^0,0) | <data type expression> | ...

Choose the data type for the output. The type can be inherited, specified directly, or expressed as a data type object such as Simulink.NumericType.

Click the **Show data type assistant** button [ ] to display the **Data Type Assistant**, which helps you set the data type attributes. For more information, see “Specify Data Types Using Data Type Assistant”.

**Programmatic Use**

**Block Parameter:** OutDataTypeStr

**Type:** character vector

**Values:** 'Inherit: Inherit via back propagation' | 'single' | 'int8' | 'uint8' | int16 | 'uint16' | 'int32' | 'uint32' | 'int64' | 'uint64' | fixdt(1,16,0) | fixdt(1,16,2^0,0) | fixdt(1,16,2^0,0) | '<data type expression>'

**Default:** 'Double'

### Lock output data type setting against changes by the fixed-point tools — Prevent fixed-point tools from overriding data types

off (default) | on

Select to lock the output data type setting of this block against changes by the Fixed-Point Tool and the Fixed-Point Advisor. For more information, see “Use Lock Output Data Type Setting” (Fixed-Point Designer).

**Programmatic Use**

**Block Parameter:** LockScale

**Type:** character vector

**Values:** 'off' | 'on'

**Default:** 'off'
Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>double</th>
<th>enumerated</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Direct Feedthrough</th>
<th>no</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multidimensional Signals</td>
<td>no</td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
</tr>
</tbody>
</table>

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

PLC Code Generation
Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

See Also
Repeating Sequence | Repeating Sequence Interpolated

Introduced before R2006a
**Reset**

Add reset port to subsystem  
**Library:** Ports & Subsystems

**Description**

A Reset block placed at the root level of a Subsystem block adds a control port to the block. When a reset trigger signal occurs on the signal connected to the port, the block states of the subsystem are reset to their initial condition.

**Parameters**

**Reset trigger type — Select the type of trigger event**  
*level (default)| rising| falling| either| level hold*

Select the type of trigger signal that resets the subsystem block states.

- **level**
  - Reset the block states when the trigger signal is nonzero at the current time step or changes from nonzero at the previous time step to zero at the current time step.

- **rising**
  - Reset the block states when the trigger signal rises from a zero to a positive value or from a negative to a positive value.

- **falling**
  - Reset the block states when the trigger signal falls from a positive value to zero or from a positive to a negative value.

- **either**
  - Reset the block states when the trigger signal changes from a zero to a nonzero value or changes sign.

- **level hold**
  - Reset the block states when the trigger signal is nonzero at the current time step.
Programmatic Use
Block Parameter: ResetTriggerType
Type: character vector
Value: 'level' | 'rising' | 'falling' | 'either' | 'level hold'
Default: 'level'

Propagate sizes of variable-size signals — Select when to propagate a variable-size signal
During execution (default) | Only when enabling

Select when to propagate a variable-size signal.

During execution
Propagate variable-size signals at each time step.

Only when resetting
Propagate variable-size signals when resetting a Subsystem block containing a Reset port block. When you select this option, sample time must be periodic.

Programmatic Use
Block Parameter: PropagateVarSize
Type: character vector
Value: 'During execution' | 'Only when resetting'
Default: 'During execution'

Enable zero-crossing detection — Control zero-crossing detection
on (default) | off

Control zero-crossing detection.

☐ on
Detect zero crossings.

☐ off
Do not detect zero crossings.

Programmatic Use
Block Parameter: ZeroCross
Type: character vector
Value: 'on' | 'off'
Default: 'on'
See Also

**Blocks**
Reset | Resettable Subsystem | Subsystem

**Topics**
“Use Resettable Subsystems”
“Simulink Block Support for Variable-Size Signals”

**Introduced in R2015a**
Reset Function

Executes contents on a model reset event
Library: Simulink / User-Defined Functions

Description

The Reset Function block is a pre-configured subsystem block that executes on a model reset event. By default, the Reset Function block includes an Event Listener block with Event set to Reset, a Constant block with Constant value set to 0, and a State Writer block.

Replace the Constant block with source blocks that generate the state value for the State Writer block.

Conditionally executed subsystem blocks with output ports fully supported within Terminate Function blocks

For a list of unsupported blocks and features, see “Initialize, Reset, and Terminate Function Limitations”.

A model can potentially have multiple Reset Function blocks with each block having a different Event name. Each of these reset events appear in the generated code as a different function.
**Block Characteristics**

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>bus</th>
<th>double</th>
<th>enumerated</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
<th>string</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>limited</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

a. Actual data type or capability support depends on block implementation.

**Extended Capabilities**

**C/C++ Code Generation**
Generate C and C++ code using Simulink® Coder™.

**Fixed-Point Conversion**
Design and simulate fixed-point systems using Fixed-Point Designer™.

**See Also**
Event Listener | Initialize Function | State Reader | State Writer | Terminate Function

**Topics**
“Using Initialize, Reset, and Terminate Functions”
“Create Test Harness to Generate Function Calls”
“Generate Code That Responds to Initialize, Reset, and Terminate Events” (Simulink Coder)
Resettable Delay

Delay input signal by variable sample period and reset with external signal

**Library:**
- Simulink / Discrete
- HDL Coder / Discrete

**Description**

The Resettable Delay block is a variant of the Delay block that has the source of the initial condition set to **Input port** and the external reset algorithm set to **Rising**, by default.

**Ports**

**Input**

- **u** — Data input signal
  - scalar | vector
  
  Input data signal delayed according to parameters settings.

  **Data Types:** single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point | enumerated | bus

- **d** — Delay length
  - scalar

  Delay length specified as inherited from an input port. Enabled when you select the **Delay length: Source** parameter as **Input port**.

  **Data Types:** single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

- **Enable** — External enable signal
  - scalar
Enable signal that enables or disables execution of the block. To create this port, select the **Show enable port** parameter.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point

**External reset — External reset signal**

scalar

External signal that resets execution of the block to the initial condition. To create this port, select the **External reset** parameter.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point

**x0 — Initial condition**

scalar | vector

Initial condition specified as inherited from an input port. Enabled when you select the **Initial Condition: Source** parameter as **Input port**.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

**Output**

**Port_1 — Output signal**

scalar | vector

Output signal that is the input signal delayed by the length of time specified by the parameter **Delay length**. The initial value of the output signal depends on several conditions. See “Initial Block Output” on page 1-360.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus
Parameters

Main

Delay length — Delay length
Dialog (default) | Input port

Specify whether to enter the delay length directly on the dialog box (fixed delay) or to inherit the delay from an input port (variable delay).

- If you set Source to Dialog, enter the delay length in the edit field under Value.
- If you set Source to Input port, verify that an upstream signal supplies a delay length for the d input port. You can also specify its maximum value by specifying the parameter Upper limit.

Specify the scalar delay length as a real, non-negative integer. An out-of-range or non-integer value in the dialog box (fixed delay) returns an error. An out-of-range value from an input port (variable delay) casts it into the range. A non-integer value from an input port (variable delay) truncates it to the integer.

Programmatic Use
Block Parameter: DelayLengthSource
Type: character vector
Values: 'Dialog' | 'Input port'
Default: 'Dialog'
Block Parameter: DelayLength
Type: character vector
Values: scalar
Default: '2'
Block Parameter: DelayLengthUpperLimit
Type: character vector
Values: scalar
Default: '100'

Initial condition — Initial condition
Input port (default) | Dialog

Specify whether to enter the initial condition directly on the dialog box or to inherit the initial condition from an input port.
• If you set **Source** to **Dialog**, enter the initial condition in the edit field under **Value**.
• If you set **Source** to **Input port**, verify that an upstream signal supplies an initial condition for the \( x_0 \) input port.

Simulink converts offline the data type of **Initial condition** to the data type of the input signal \( u \) using a round-to-nearest operation and saturation.

**Note** When **State name must resolve to Simulink signal object** is selected on the **State Attributes** pane, the block copies the initial value of the signal object to the **Initial condition** parameter. However, when the source for **Initial condition** is **Input port**, the block ignores the initial value of the signal object.

**Programmatic Use**

**Block Parameter**: InitialConditionSource  
**Type**: character vector  
**Values**: 'Dialog' | 'Input port' |  
**Default**: 'Input Port'

**Block Parameter**: InitialCondition  
**Type**: character vector  
**Values**: scalar  
**Default**: ''

**Input processing — Specify sample- or frame-based processing**

Elements as channels (sample based) (default) | Columns as channels (frame based)

Specify whether the block performs sample- or frame-based processing:

• **Columns as channels (frame based)** — Treat each column of the input as a separate channel (frame-based processing).

**Note** Frame-based processing requires a DSP System Toolbox license.

For more information, see “Sample- and Frame-Based Concepts” (DSP System Toolbox).

• **Elements as channels (sample based)** — Treat each element of the input as a separate channel (sample-based processing).
Use **Input processing** to specify whether the block performs sample- or frame-based processing. The block accepts frame-based signals for the input \( u \). All other input signals must be sample-based.

<table>
<thead>
<tr>
<th>Input Signal ( u )</th>
<th>Input Processing Mode</th>
<th>Block Works?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sample based</td>
<td>Sample based</td>
<td>Yes</td>
</tr>
<tr>
<td>Frame based</td>
<td></td>
<td>No, produces an error</td>
</tr>
<tr>
<td>Sample based</td>
<td>Frame based</td>
<td>Yes</td>
</tr>
<tr>
<td>Frame based</td>
<td></td>
<td>Yes</td>
</tr>
</tbody>
</table>

For more information about these two processing modes, see “Sample- and Frame-Based Concepts” (DSP System Toolbox).

**Programmatic Use**

**Block Parameter**: `InputProcessing`  
**Type**: character vector  
**Values**: `'Columns as channels (frame based)'` | `'Elements as channels (sample based)'`  
**Default**: `'Elements as channels (sample based)'`

**Use circular buffer for state — Circular buffer for storing state**

`off` (default) | `on`

Select to use a circular buffer for storing the state in simulation and code generation. Otherwise, an array buffer stores the state.

Using a circular buffer can improve execution speed when the delay length is large. For an array buffer, the number of copy operations increases as the delay length goes up. For a circular buffer, the number of copy operations is constant for increasing delay length.

If one of the following conditions is true, an array buffer always stores the state because a circular buffer does not improve execution speed:

- For sample-based signals, the delay length is 1.
- For frame-based signals, the delay length is no larger than the frame size.

**Programmatic Use**

**Block Parameter**: `UseCircularBuffer`  
**Type**: character vector  
**Values**: `'off' | 'on'`
**Default:** 'off'

**Prevent direct feedthrough — Prevent direct feedthrough**

off (default) | on

Select to increase the delay length from zero to the lower limit for the *Input processing* mode:

- For sample-based signals, increase the minimum delay length to 1.
- For frame-based signals, increase the minimum delay length to the frame length.

Selecting this check box prevents direct feedthrough from the input port, u, to the output port. However, this check box cannot prevent direct feedthrough from the initial condition port, x0, to the output port.

**Dependency**

To enable this parameter, set *Delay length: Source* to Input port.

**Programmatic Use**

**Block Parameter:** PreventDirectFeedthrough  
**Type:** character vector  
**Values:** 'off' | 'on'  
**Default:** 'off'

**Remove delay length check in generated code — Remove delay length out-of-range check**

off (default) | on

Select to remove code that checks for out-of-range delay length.

<table>
<thead>
<tr>
<th>Check Box</th>
<th>Result</th>
<th>When to Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selected</td>
<td>Generated code does not include conditional statements to check for out-of-range delay length.</td>
<td>For code efficiency</td>
</tr>
<tr>
<td>Cleared</td>
<td>Generated code includes conditional statements to check for out-of-range delay length.</td>
<td>For safety-critical applications</td>
</tr>
</tbody>
</table>
Dependency

To enable this parameter, set \textbf{Delay length: Source} to \textit{Input port}.

Programmatic Use

\textbf{Block Parameter:} RemoveDelayLengthCheckInGeneratedCode  
\textbf{Type:} character vector  
\textbf{Values:} 'off' | 'on'  
\textbf{Default:} 'off'

\textbf{Diagnostic for delay length — Diagnostic checks for delay length}

\textbf{None} (default) | Warning | Error

Specify whether to produce a warning or error when the input \textit{d} is less than the lower limit or greater than the \textbf{Delay length: Upper limit}. The lower limit depends on the setting for \textbf{Prevent direct feedthrough}.

- If the check box is cleared, the lower limit is zero.
- If the check box is selected, the lower limit is 1 for sample-based signals and frame length for frame-based signals.

Options for the diagnostic include:

- \textbf{None} — Simulink software takes no action.
- \textbf{Warning} — Simulink software displays a warning and continues the simulation.
- \textbf{Error} — Simulink software terminates the simulation and displays an error.

Dependency

To enable this parameter, set \textbf{Delay length: Source} to \textit{Input port}.

Programmatic Use

\textbf{Block Parameter:} DiagnosticForDelayLength  
\textbf{Type:} character vector  
\textbf{Values:} 'None' | 'Warning' | 'Error'  
\textbf{Default:} 'None'

\textbf{Show enable port — Create enable port}

\textbf{off} (default) | on

Select to control execution of this block with an enable port. The block is considered enabled when the input to this port is nonzero, and is disabled when the input is 0. The value of the input is checked at the same time step as the block execution.
**Programmatic Use**

**Block Parameter:** `ShowEnablePort`  
**Type:** character vector  
**Values:** 'off' | 'on'  
**Default:** 'off'

**External reset — External state reset**

Specify the trigger event to use to reset the states to the initial conditions.

<table>
<thead>
<tr>
<th>Reset Mode</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>No reset.</td>
</tr>
<tr>
<td>Rising</td>
<td>Reset on a rising edge.</td>
</tr>
<tr>
<td>Falling</td>
<td>Reset on a falling edge.</td>
</tr>
<tr>
<td>Either</td>
<td>Reset on either a rising or falling edge.</td>
</tr>
<tr>
<td>Level</td>
<td>Reset in either of these cases:</td>
</tr>
<tr>
<td></td>
<td>• when the reset signal is nonzero at the current time step</td>
</tr>
<tr>
<td></td>
<td>• when the reset signal value changes from nonzero at the previous time step to zero at the current time step</td>
</tr>
<tr>
<td>Level hold</td>
<td>Reset when the reset signal is nonzero at the current time step</td>
</tr>
</tbody>
</table>

The reset signal must be a scalar of type `single`, `double`, `boolean`, or `integer`. Fixed point data types, except for `ufix1`, are not supported.

**Programmatic Use**

**Block Parameter:** `ExternalReset`  
**Type:** character vector  
**Values:** 'None' | 'Rising' | 'Falling' | 'Either' | 'Level' | 'Level hold'  
**Default:** 'Rising'

**Sample time (-1 for inherited) — Discrete interval between sample time hits**

-1 (default) | scalar
Specify the time interval between samples. To inherit the sample time, set this parameter to -1. This block supports discrete sample time, but not continuous sample time.

**Programmatic Use**

**Block Parameter:** SampleTime

**Type:** character vector  
**Value:** real scalar  
**Default:** '-1'

**State Attributes**

**State name — Unique name for block state**

'' (default) | alphanumeric string

Use this parameter to assign a unique name to the block state. The default is ''. When this field is blank, no name is assigned. When using this parameter, remember these considerations:

- A valid identifier starts with an alphabetic or underscore character, followed by alphanumeric or underscore characters.
- The state name applies only to the selected block.

This parameter enables **State name must resolve to Simulink signal object** when you click **Apply**.

For more information, see “Apply Storage Classes to Individual Signal, State, and Parameter Data Elements” (Simulink Coder).

**Programmatic Use**

**Block Parameter:** StateName

**Type:** character vector  
**Values:** unique name  
**Default:** ''

**State name must resolve to Simulink signal object — Require state name resolve to a signal object**

off (default) | on

Select this check box to require that the state name resolves to a Simulink signal object.
Dependencies

To enable this parameter, specify a value for State name. This parameter appears only if you set the model configuration parameter Signal resolution to a value other than None.

Selecting this check box disables Code generation storage class.

Programmatic Use

Block Parameter: StateMustResolveToSignalObject

Type: character vector

Values: 'off' | 'on'

Default: 'off'

Signal object class — Custom storage class package name

Simulink.Signal (default)|<StorageClass.PackageName>

Choose a custom storage class package by selecting a signal object class that the target package defines. For example, to apply custom storage classes from the built-in package mpt, select mpt.Signal. Unless you use an ERT-based code generation target with Embedded Coder, custom storage classes do not affect the generated code.

If the class that you want does not appear in the list, select Customize class lists.

For instructions, see “Target Class Does Not Appear in List of Signal Object Classes” (Embedded Coder).

For information about storage classes, see “Apply Storage Classes to Individual Signal, State, and Parameter Data Elements” (Simulink Coder). For information about custom storage classes, see “Apply Built-In and Customized Storage Classes to Data Elements” (Embedded Coder).

Programmatic Use

Block Parameter: StateSignalObject

Type: character vector

Values: 'Simulink.Signal' | '<StorageClass.PackageName>'

Default: 'Simulink.Signal'

Code generation storage class — State storage class for code generation

Auto (default)|Model default|ExportedGlobal|ImportedExtern|
ImportedExternPointer|BitField (Custom)|Model default|ExportToFile (Custom)|ImportFromFile (Custom)|FileScope (Custom)|AutoScope (Custom)|Struct (Custom)|GetSet (Custom)|Reusable (Custom)

Select state storage class for code generation.
• **Auto** is the appropriate storage class for states that you do not need to interface to external code.

• **StorageClass** applies the storage class or custom storage class that you select from the list. For information about storage classes, see “Apply Storage Classes to Individual Signal, State, and Parameter Data Elements” (Simulink Coder). For information about custom storage classes, see “Apply Built-In and Customized Storage Classes to Data Elements” (Embedded Coder).

Use **Signal object class** to select custom storage classes from a package other than Simulink.

**Dependencies**

To enable this parameter, specify a value for **State name**.

**Programmatic Use**

**Block Parameter:** StateStorageClass  
**Type:** character vector  
**Values:** 'Auto' | 'Model default' | 'ExportedGlobal' | 'ImportedExtern' | 'ImportedExternPointer' | 'Custom' | ...  
**Default:** 'Auto'

**Block Characteristics**

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>bus</th>
<th>double</th>
<th>enumerated</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Direct Feedthrough</strong></td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Multidimensional Signals</strong></td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Variable-Size Signals</strong></td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Zero-Crossing Detection</strong></td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

Consider using the Model Discretizer to map these continuous blocks into discrete equivalents that support code generation. From a model, in the Apps tab, under Control Systems, click Model Discretizer.

Not recommended for production code.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

For information about HDL code generation, see the “HDL Code Generation” on page 1-373 section on the Delay page.

Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

See Also
Delay | Unit Delay | Variable Integer Delay

Topics
“Using Enabled Subsystems”

Introduced in R2012b
**Resettable Subsystem**

Subsystem whose block states reset with external trigger  
**Library:** Simulink / Ports & Subsystems

**Description**

The Resettable Subsystem block is a Subsystem block preconfigured as a starting point for creating a subsystem that resets the block states each time the control port receives a trigger signal.

**Ports**

**Input**

In — **Signal input to a subsystem block**  
scalar | vector | matrix

Placing an Inport block in a Subsystem block adds an external input port to the block. The port label matches the name of the Inport block.
Use Inport blocks to receive signals from the local environment.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus

**Reset — Control signal input to a subsystem block**
scalar | vector | matrix

Placing a Reset block in a Subsystem block adds an external input port to the block and changes the block to a Resettable Subsystem block.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point

**Output**

**Out — Signal output from a subsystem**
scalar | vector | matrix

Placing an Outport block in a Subsystem block adds an output port from the block. The port label on the Subsystem block is the name of the Outport block.

Use Outport blocks to send signals to the local environment.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus

**Block Characteristics**

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean$^a$</th>
<th>bus$^a$</th>
<th>double$^a$</th>
<th>enumerated$^a$</th>
<th>fixed point$^a$</th>
<th>integer$^a$</th>
<th>single$^a$</th>
<th>string$^a$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>limited$^a$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>limited$^a$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Zero-Crossing Detection | no
---|---
a. Actual data type or capability support depends on block implementation.

### Extended Capabilities

**C/C++ Code Generation**
Generate C and C++ code using Simulink® Coder™.

Actual code generation support depends on block implementation.

**Fixed-Point Conversion**
Design and simulate fixed-point systems using Fixed-Point Designer™.

Actual data type support depends on block implementation.

### See Also

*Blocks*
*Reset* | *Subsystem*

### Topics

“Use Resettable Subsystems”

**Introduced in R2015a**
Reshape

Change dimensionality of signal

Library: Simulink / Math Operations
HDL Coder / Math Operations

Description

The Reshape block changes the dimensionality of the input signal to a dimensionality that you specify, using the **Output dimensionality** parameter. For example, you can use the block to change an $N$-element vector to a 1-by-$N$ or $N$-by-1 matrix signal.

Ports

Input

**Port_1 — Input signal**
scalar | vector | matrix

Input signal whose dimensions are changed based on the **Output dimensionality** parameter.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus

Output

**Port_1 — Output signal**
scalar | vector | matrix

Output signal created with the dimensions specified in the **Output dimensionality** parameter.
Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 |
Boolean | fixed point | enumerated | bus

Parameters

Output dimensionality — Dimensions of output signal
1-D array (default) | Column vector (2-D) | Row vector (2-D) | Customize |
Derive from reference input port

Specify the dimensionality of the output signal.

<table>
<thead>
<tr>
<th>Output Dimensionality</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-D array</td>
<td>Converts a multidimensional array to a vector (1-D array) array signal. The output vector consists of the first column of the input matrix followed by the second column, and so on. (This option leaves a vector input unchanged.)</td>
</tr>
<tr>
<td>Column vector</td>
<td>Converts a vector, matrix, or multidimensional input signal to a column matrix, a $M$-by-$1$ matrix, where $M$ is the number of elements in the input signal. For matrices, the conversion is done in column-major order. For multidimensional arrays, the conversion is done along the first dimension.</td>
</tr>
<tr>
<td>Row vector</td>
<td>Converts a vector, matrix, or multidimensional input signal to a row matrix, a $1$-by-$N$ matrix where $N$ is the number of elements in the input signal. For matrices, the conversion is done in column-major order. For multidimensional arrays, the conversion is done along the first dimension.</td>
</tr>
<tr>
<td>Customize</td>
<td>Converts the input signal to an output signal whose dimensions you specify, using the <strong>Output dimensions</strong> parameter.</td>
</tr>
<tr>
<td>Derive from reference input port</td>
<td>Creates a second input port, <strong>Ref</strong>, on the block. Derives the dimensions of the output signal from the dimensions of the signal input to the Ref input port. Selecting this option disables the <strong>Output dimensions</strong> parameter. When you select this parameter, the input signals for both inport ports, <strong>U</strong> and <strong>Ref</strong>, must have the same sampling mode (sample-based or frame-based).</td>
</tr>
</tbody>
</table>
Programmatic Use
Block Parameter: OutputDimensionality
Type: character vector
Value: '1-D array' | 'Column vector (2-D)' | 'Row vector (2-D)' |
'Customize' | 'Derive from reference input port'
Default: '1-D array'

Output dimensions — Custom dimensions of output signal
[1,1] (default) | [integer] | [integer,integer]

Specify the dimensions for the output signal. The value can be a one- or multi-element vector. A value of [N] outputs a vector of size N. A value of [M N] outputs an M-by-N matrix. The number of elements of the input signal must match the number of elements specified by the Output dimensions parameter. For multidimensional arrays, the conversion is done along the first dimension.

Dependency
To enable this parameter, set Output dimensionality to Customize.

Programmatic Use
Block Parameter: OutputDimensions
Type: character vector
Value: ' [integer,integer]' |
Default: ' [1,1]' 

Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>bus</th>
<th>double</th>
<th>enumerated</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Zero-Crossing Detection

| Zero-Crossing Detection | no |

---

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

### HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

#### HDL Architecture

This block has a single, default HDL architecture.

#### HDL Block Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConstrainedOutputPipeline</td>
<td>Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. For more details, see “ConstrainedOutputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td>InputPipeline</td>
<td>Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “InputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td>OutputPipeline</td>
<td>Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “OutputPipeline” (HDL Coder).</td>
</tr>
</tbody>
</table>
Complex Data Support
This block supports code generation for complex signals.

PLC Code Generation
Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

See Also
Squeeze

Topics
“Combine Buses into an Array of Buses”

Introduced before R2006a
Rocker Switch

Toggle parameter between two values

Library: Simulink / Dashboard

Description

The Rocker Switch block toggles the value of the connected block parameter between two values during simulation. For example, you can connect the Rocker Switch block to a Switch block in your model and change its state during simulation. Use the Rocker Switch block with other Dashboard blocks to create an interactive dashboard for your model.

Double-clicking the Rocker Switch block does not open its dialog box during simulation and when the block is selected. To edit the block’s parameters, you can use the Property Inspector, or you can right-click the block and select Block Parameters from the context menu.

Connecting Dashboard Blocks

Dashboard blocks do not use ports to make connections. To connect Dashboard blocks to variables and block parameters in your model, use connect mode. Connect mode facilitates the process of connecting Dashboard blocks in your model, especially when you want to connect multiple blocks at once. If you only want to connect a single Dashboard block, you can also use the Connection table in the block dialog box to make the connection.

Tip To facilitate understanding and debugging your model, you can modify Dashboard block connections in your model during normal and accelerator mode simulations.
**Note** Dashboard blocks cannot connect to variables until you update your model diagram. To connect Dashboard blocks to variables or modify variable values between opening your model and running a simulation, update your model diagram using **Ctrl+D**.

To enter connect mode, click the **Connect** button that appears above your unconnected Dashboard block when you pause on it.

In connect mode, when you select one or more signals or blocks, a list of parameters and signals available for connection appears. Select a signal or parameter from the list to connect the selected Dashboard block. To connect another Dashboard block, pause on the block and click the **Connect** button above it. Then, make a selection of signals and blocks in your model, and choose a signal or parameter to connect.
When you finish connecting the Dashboard blocks in your model, click the Exit button in the upper-right of the canvas to exit connect mode.

**Tip** You can hide the message shown on unconnected blocks using the `set_param` function with the `ShowInitialText` block parameter.

**Parameter Logging**

Tunable parameters connected to Dashboard blocks are logged to the Simulation Data Inspector, where you can view parameter values along with logged signal data. You can access logged parameter data in the MATLAB workspace by exporting the parameter data from the Simulation Data Inspector UI or by using the `Simulink.sdi.exportRun` function. For more information about exporting data with the Simulation Data Inspector UI, see “Export Data from the Simulation Data Inspector”. The parameter data is stored in a `Simulink.SimulationData.Parameter` object, accessible as an element in the exported `Simulink.SimulationData.Dataset`. 
Limitations

- Dashboard blocks can only connect to real scalar signals.
- You cannot use the Connection table to connect a Dashboard block to a block that is commented out. When you connect a Dashboard block to a commented block using connect mode, the Dashboard block does not display the connected value until you uncomment the block.
- Dashboard blocks cannot connect to signals inside referenced models.
- When you simulate a model hierarchy, Dashboard blocks inside referenced models do not update.
- Dashboard blocks do not support rapid accelerator simulation.
- Some signals are not available during accelerator mode simulations. To view such signals on a Dashboard block, mark the signal for logging.
- Parameters specified by indexing a variable array do not appear in the Connection table. For example, a block parameter defined as `engine(1)` using the variable `engine` does not appear in the table.

To access the parameter in the Connection table, assign the indexed value to a scalar variable, such as `engine_1`. Then, use the scalar variable to define the block parameter.

- When you connect a Dashboard block to a variable or parameter during simulation, the data for that variable or parameter does not get logged to the Simulation Data Inspector. To log variable and parameter data to the Simulation Data Inspector, connect the Dashboard block to the variable or parameter prior to simulation.
- When you simulate a model in external mode with the Default parameter behavior set to Inlined, Dashboard blocks can appear to change parameter and variable values. However, the change does not propagate to the simulation. For example, the Gain block displays changes made to its Gain parameter using the Dashboard blocks, but the Gain value used in the simulation does not change.

Parameters

Connection — Select a variable or block parameter to connect variable and parameter connection options

Select the variable or block parameter to control using the Connection table. Populate the Connection table by selecting one or more blocks in your model. Select the radio
button next to the variable or parameter you want to control, and click **Apply**. To facilitate understanding and debugging your model, you can connect Dashboard blocks to variables and parameters in your model during simulation.

**Note** To see workspace variables in the connection table, update the model diagram using **Ctrl+D**.

---

**Programmatic Use**

To programmatically connect a Dashboard block to a tunable parameter or a variable, use a `Simulink.HMI.ParamSourceInfo` object. The `Simulink.HMI.ParamSourceInfo` object contains four properties. Some of the properties apply to connecting Dashboard blocks to parameters, and some apply to connecting Dashboard blocks to variables. Not all fields have a value for a connection because a given Dashboard block connects to either a parameter or a variable.

**Block Parameter:** Binding  
**Type:** `Simulink.HMI.ParamSourceInfo`  
**Default:** `[]`

**States — Pair values and labels**

Scalar and character vector

Pairs of values to assign to the connected variable or parameter and text to display on the block. Switches have two states — **Top** and **Bottom** — one corresponding to each switch position. Each state contains a **Value** and a **Label**.

- **Value** — Value to assign to the connected variable or parameter when the switch is in the corresponding position.
- **Label** — Text to display on the block for the corresponding position.

This table describes the default configuration for the block.

**States**

<table>
<thead>
<tr>
<th>Position</th>
<th>State Value</th>
<th>State Label</th>
</tr>
</thead>
<tbody>
<tr>
<td>Top</td>
<td>0</td>
<td>on</td>
</tr>
<tr>
<td>Bottom</td>
<td>1</td>
<td>off</td>
</tr>
</tbody>
</table>
Programmatic Use

To configure the States for the block programmatically, specify the value of the States parameter as a structure array containing two elements with fields:

- Value — Scalar double value for the state.
- Label — String or character array to use as the label for the switch position.

```matlab
topState.Value = 0;
topState.Label = 'Off';
bottomState.Value = 1;
bottomState.Label = 'On';
switchStates = [topState bottomState];
```

**Block Parameter: States**

**Type:** two element array of structures

**Label — Block label position**

Top (default) | Bottom | Hide

Position of the block label. When the block is connected to an element in the model, the label is the name of the connected element.

**Programmatic Use**

**Block Parameter:** LabelPosition

**Type:** character vector

**Values:** 'Top' | 'Bottom' | 'Hide'

**Default:** 'Top'

---

**Block Characteristics**

<table>
<thead>
<tr>
<th>Data Types</th>
<th>double</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>no</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Zero-Crossing Detection | no

### Extended Capabilities

**C/C++ Code Generation**
Generate C and C++ code using Simulink® Coder™.

Dashboard blocks are ignored for code generation.

### See Also
Rotary Switch | Slider Switch | Toggle Switch

### Topics
“Tune and Visualize Your Model with Dashboard Blocks”
“Decide How to Visualize Simulation Data”

**Introduced in R2015a**
Rotary Switch

Switch parameter to set values on dial
Library: Simulink / Dashboard

Description

The Rotary Switch changes the value of the connected block parameter to several specified values during simulation. For example, you can connect the Rotary Switch block to the amplitude or frequency of an input signal in your model and change its characteristics during simulation. Use the Rotary Switch block with other Dashboard blocks to create an interactive dashboard to control your model.

Double-clicking the Rotary Switch block does not open its dialog box during simulation and when the block is selected. To edit the block's parameters, you can use the Property Inspector, or you can right-click the block and select Block Parameters from the context menu.

Connecting Dashboard Blocks

Dashboard blocks do not use ports to make connections. To connect Dashboard blocks to variables and block parameters in your model, use connect mode. Connect mode facilitates the process of connecting Dashboard blocks in your model, especially when you want to connect multiple blocks at once. If you only want to connect a single Dashboard block, you can also use the Connection table in the block dialog box to make the connection.

Tip To facilitate understanding and debugging your model, you can modify Dashboard block connections in your model during normal and accelerator mode simulations.
Note Dashboard blocks cannot connect to variables until you update your model diagram. To connect Dashboard blocks to variables or modify variable values between opening your model and running a simulation, update your model diagram using Ctrl+D.

To enter connect mode, click the Connect button that appears above your unconnected Dashboard block when you pause on it.

In connect mode, when you select one or more signals or blocks, a list of parameters and signals available for connection appears. Select a signal or parameter from the list to connect the selected Dashboard block. To connect another Dashboard block, pause on the block and click the Connect button above it. Then, make a selection of signals and blocks in your model, and choose a signal or parameter to connect.
When you finish connecting the Dashboard blocks in your model, click the **Exit** button in the upper-right of the canvas to exit connect mode.

**Tip** You can hide the message shown on unconnected blocks using the `set_param` function with the `ShowInitialText` block parameter.

**Parameter Logging**

Tunable parameters connected to Dashboard blocks are logged to the Simulation Data Inspector, where you can view parameter values along with logged signal data. You can access logged parameter data in the MATLAB workspace by exporting the parameter data from the Simulation Data Inspector UI or by using the `Simulink.sdi.exportRun` function. For more information about exporting data with the Simulation Data Inspector UI, see “Export Data from the Simulation Data Inspector”. The parameter data is stored in a `Simulink.SimulationData.Parameter` object, accessible as an element in the exported `Simulink.SimulationData.Dataset`. 

1-1693
Limitations

- Dashboard blocks can only connect to real scalar signals.
- You cannot use the Connection table to connect a Dashboard block to a block that is commented out. When you connect a Dashboard block to a commented block using connect mode, the Dashboard block does not display the connected value until you uncomment the block.
- Dashboard blocks cannot connect to signals inside referenced models.
- When you simulate a model hierarchy, Dashboard blocks inside referenced models do not update.
- Dashboard blocks do not support rapid accelerator simulation.
- Some signals are not available during accelerator mode simulations. To view such signals on a Dashboard block, mark the signal for logging.
- Parameters specified by indexing a variable array do not appear in the Connection table. For example, a block parameter defined as engine(1) using the variable engine does not appear in the table.

To access the parameter in the Connection table, assign the indexed value to a scalar variable, such as engine_1. Then, use the scalar variable to define the block parameter.

- When you connect a Dashboard block to a variable or parameter during simulation, the data for that variable or parameter does not get logged to the Simulation Data Inspector. To log variable and parameter data to the Simulation Data Inspector, connect the Dashboard block to the variable or parameter prior to simulation.
- When you simulate a model in external mode with the Default parameter behavior set to Inlined, Dashboard blocks can appear to change parameter and variable values. However, the change does not propagate to the simulation. For example, the Gain block displays changes made to its Gain parameter using the Dashboard blocks, but the Gain value used in the simulation does not change.

Parameters

Connection — Select a variable or block parameter to connect
variable and parameter connection options

Select the variable or block parameter to control using the Connection table. Populate the Connection table by selecting one or more blocks in your model. Select the radio
button next to the variable or parameter you want to control, and click **Apply**. To facilitate understanding and debugging your model, you can connect Dashboard blocks to variables and parameters in your model during simulation.

**Note** To see workspace variables in the connection table, update the model diagram using **Ctrl+D**.

**Programmatic Use**

To programmatically connect a Dashboard block to a tunable parameter or a variable, use a `Simulink.HMI.ParamSourceInfo` object. The `Simulink.HMI.ParamSourceInfo` object contains four properties. Some of the properties apply to connecting Dashboard blocks to parameters, and some apply to connecting Dashboard blocks to variables. Not all fields have a value for a connection because a given Dashboard block connects to either a parameter or a variable.

**Block Parameter:** Binding  
**Type:** `Simulink.HMI.ParamSourceInfo`  
**Default:** `[]`

**States — Position labels and values**

`scalar and character vector`

Pairs of numeric values to assign to the connected variable or parameter with labels for the block. A State consists of a **State Value** and a **State Label**.

- **State Value** — Value assigned to the connected variable or parameter when the Rotary Switch block is positioned at the corresponding **State Label**.
- **State Label** — Label for the switch position. You can use the **State Label** to display the value the connected variable or parameter takes when the Rotary Switch block points to the **State Label**, or you can use a descriptive text label.

Click the + button to add **States**.

The default configuration for the block includes these **States**:
**States**

<table>
<thead>
<tr>
<th>State Value</th>
<th>State Label</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Off</td>
</tr>
<tr>
<td>1</td>
<td>Low</td>
</tr>
<tr>
<td>2</td>
<td>Medium</td>
</tr>
<tr>
<td>3</td>
<td>High</td>
</tr>
</tbody>
</table>

**Programmatic Use**

To configure the **States** for the block programmatically, specify the value of the **States** parameter as a structure array with fields:

- **Value** — Scalar double value for the state.
- **Label** — String or character array to use as the label for the switch position.

Include a structure element in the array for each state you want to specify for the block.

```matlab
state1.Value = 1;
state1.Label = 'State 1';
state2.Value = 2;
state2.Label = 'State 2';
states = [state1 state2];
```

**Block Parameter:** States  
**Type:** structure array

**Enumerated Data Type — Specify state values and labels using an enumerated data type**  
*off* (default) | *on*

You can use an enumerated data type that pairs a numeric value with each enumeration to configure the state values and labels for the block. To specify the states for the block using an enumerated data type, first select the **Enumerated Data Type** option. Then, specify the name of the enumerated data type in the text box. The definition for the specified enumerated data type must be saved on the MATLAB path or in the base workspace.

**Example:** `myEnumType`
**Programmatic Use**

To programatically specify the state labels and values for the block using an enumerated data type, specify 'on' for the UseEnumeratedDataType parameter and the name of the enumerated data type for the EnumeratedDataType parameter.

**Block Parameter:** UseEnumeratedDataType

- **Type:** string or character array
- **Values:** 'on' | 'off'
- **Default:** 'off'

**Block Parameter:** EnumeratedDataType

- **Type:** string or character array
- **Default:** ''

**Label — Block label position**

- **Top** (default) | **Bottom** | **Hide**

Position of the block label. When the block is connected to an element in the model, the label is the name of the connected element.

**Programmatic Use**

**Block Parameter:** LabelPosition

- **Type:** character vector
- **Values:** 'Top' | 'Bottom' | 'Hide'
- **Default:** 'Top'

**Block Characteristics**

<table>
<thead>
<tr>
<th>Data Types</th>
<th>double</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>no</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

Dashboard blocks are ignored for code generation.

See Also
Knob | Rocker Switch | Slider | Slider Switch | Toggle Switch

Topics
“Tune and Visualize Your Model with Dashboard Blocks”
“Decide How to Visualize Simulation Data”

Introduced in R2015a
**Rounding Function**

Apply rounding function to signal  
**Library:** Simulink / Math Operations

**Description**

The Rounding Function block rounds each element of the input signal to produce the output signal.

You select the type of rounding from the **Function** parameter list. The name of the selected function appears on the block.

**Tip** Use the Rounding Function block when you want vector or matrix output.

**Ports**

**Input**

**Port_1 — Input signal**  
scalar | vector | matrix

Input signal to which the rounding function is applied.

Data Types: single | double

**Output**

**Port_1 — Output signal**  
scalar | vector | matrix
Output signal after the rounding function is applied to the input signal. The output signal has the same dimensions and data type as the input. Each element of the output signal is the result of applying the selected rounding function to the corresponding element of the input signal.

**Data Types:** single | double

### Parameters

**Function — Rounding function**

floor (default) | ceil | round | fix

Choose the rounding function applied to the input signal.

<table>
<thead>
<tr>
<th>Rounding function</th>
<th>Rounds each element of the input signal</th>
</tr>
</thead>
<tbody>
<tr>
<td>floor</td>
<td>To the nearest integer value towards minus infinity</td>
</tr>
<tr>
<td>ceil</td>
<td>To the nearest integer towards positive infinity</td>
</tr>
<tr>
<td>round</td>
<td>To the nearest integer</td>
</tr>
<tr>
<td>fix</td>
<td>To the nearest integer towards zero</td>
</tr>
</tbody>
</table>

**Programmatic Use**

**Block Parameter:** Operator

**Type:** character vector

**Values:** 'floor' | 'ceil' | 'round' | 'fix'

**Default:** 'floor'

**Sample time — Specify sample time as a value other than -1**

-1 (default) | scalar | vector

Specify the sample time as a value other than -1. For more information, see “Specify Sample Time”.

**Dependencies**

This parameter is not visible unless it is explicitly set to a value other than -1. To learn more, see “Blocks for Which Sample Time Is Not Recommended”.
Programmatic Use
Block Parameter: SampleTime
Type: character vector
Values: scalar or vector
Default: '-1'

Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>double</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
</tr>
</tbody>
</table>

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Code Generation Requirements
HDL code generation for the block requires that you use single data types as inputs to the block, and enable the native floating point mode. In the Configuration Parameters dialog
box, on the **HDL Code Generation > Floating Point** pane, for **Library**, select **Native Floating Point**. To learn more about using the native floating-point mode, see “Getting Started with HDL Coder Native Floating-Point Support” (HDL Coder) and “Generate Target-Independent HDL Code with Native Floating-Point” (HDL Coder).

**HDL Architecture**

This block has a single, default HDL architecture.

**HDL Block Properties**

<table>
<thead>
<tr>
<th>General</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ConstrainedOutputPipeline</strong></td>
</tr>
<tr>
<td>Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. For more details, see “ConstrainedOutputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td><strong>InputPipeline</strong></td>
</tr>
<tr>
<td>Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “InputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td><strong>OutputPipeline</strong></td>
</tr>
<tr>
<td>Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “OutputPipeline” (HDL Coder).</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Native Floating Point</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>LatencyStrategy</strong></td>
</tr>
<tr>
<td>Specify whether to map the blocks in your design to inherit, Max, Min, Zero, or Custom for the floating-point operator. The default is inherit. See also “LatencyStrategy” (HDL Coder).</td>
</tr>
<tr>
<td><strong>NFPCustomLatency</strong></td>
</tr>
<tr>
<td>To specify a value, set <strong>LatencyStrategy</strong> to Custom. HDL Coder adds latency equal to the value that you specify for the <strong>NFPCustomLatency</strong> setting. See also “NFPCustomLatency” (HDL Coder).</td>
</tr>
</tbody>
</table>

**Complex Data Support**

This block supports code generation for complex signals.
PLC Code Generation
Generate Structured Text code using Simulink® PLC Coder™.

See Also

Introduced before R2006a
Saturation

Limit input signal to the upper and lower saturation values

**Library:**
- Simulink / Commonly Used Blocks
- Simulink / Discontinuities
- HDL Coder / Discontinuities

**Description**

The Saturation block produces an output signal that is the value of the input signal bounded to the upper and lower saturation values. The upper and lower limits are specified by the parameters **Upper limit** and **Lower limit**.

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lower limit ≤ Input value ≤ Upper limit</td>
<td>Input value</td>
</tr>
<tr>
<td>Input value &lt; Lower limit</td>
<td>Lower limit</td>
</tr>
<tr>
<td>Input value &gt; Upper limit</td>
<td>Upper limit</td>
</tr>
</tbody>
</table>

**Ports**

**Input**

**Port_1 — Input signal**

Scalar | Vector

The input signal to the saturation algorithm.

Data Types: double | single | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point
Output

Port_1 — Output signal
scalar | vector

Output signal that is the value of the input signal, upper saturation limit, or lower saturation limit.

Data Types: double | single | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

Parameters

Main

Upper limit — Upper saturation boundary for the input signal
0.5 (default) | scalar | vector

Specify the upper bound on the input signal. If the input signal is greater than this boundary, then the output signal is set to this saturation value. The Upper limit parameter is converted to the output data type using round-to-nearest and saturation. Upper limit must be greater than the Output minimum parameter and less than the Output maximum parameter.

Programmatic Use
Block Parameter: UpperLimit
Type: character vector
Value: real scalar or vector
Default: '0.5'

Lower limit — Lower saturation boundary for the input signal
-0.5 (default) | scalar | vector

Specify the lower bound on the input signal. If the input signal is less than this boundary, then the output signal is set to this saturation value. The Lower limit parameter is converted to the output data type using round-to-nearest and saturation. Lower limit must be greater than the Output minimum parameter and less than the Output maximum parameter.
Programmatic Use
Block Parameter: LowerLimit
Type: character vector
Value: real scalar or vector
Default: '-0.5'

Treat as gain when linearizing — Specify the gain value
On (default) | Boolean

Select this check box to cause the commands to treat the gain as 1. The linearization commands in Simulink software treat this block as a gain in state space. Clear the box to have the commands treat the gain as 0.

Programmatic Use
Block Parameter: LinearizeAsGain
Type: character vector
Values: 'off' | 'on'
Default: 'on'

Enable zero-crossing detection — Enable zero-crossing detection
on (default) | Boolean

Select to enable zero-crossing detection. For more information, see “Zero-Crossing Detection” in the Simulink documentation.

Programmatic Use
Block Parameter: ZeroCross
Type: character vector, string
Values: 'off' | 'on'
Default: 'on'

Sample time — Specify sample time as a value other than -1
-1 (default) | scalar | vector

Specify the sample time as a value other than -1. For more information, see “Specify Sample Time”.

Dependencies

This parameter is not visible unless it is explicitly set to a value other than -1. To learn more, see “Blocks for Which Sample Time Is Not Recommended”.
**Signal Attributes**

Click the **Show data type assistant** button to display the **Data Type Assistant**, which helps you set the data type attributes. For more information, see “Specify Data Types Using Data Type Assistant”.

**Output minimum — Minimum output value for range checking**

[] (default) | scalar

Lower value of the output range that Simulink checks.

Simulink uses the minimum to perform:

- Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters”) for some blocks.
- Simulation range checking (see “Signal Ranges” and “Enable Simulation Range Checking”).
- Automatic scaling of fixed-point data types.
- Optimization of the code that you generate from the model. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. For more information, see “Optimize using the specified minimum and maximum values” (Simulink Coder).

**Note** Output minimum does not saturate or clip the actual output signal. Use the Saturation block instead.

**Programmatic Use**

**Block Parameter:** `OutMin`

**Type:** character vector

**Values:** '[]' | scalar

**Default:** '[ ]'
Output maximum — Maximum output value for range checking

[] (default) | scalar

Upper value of the output range that Simulink checks.

Simulink uses the maximum value to perform:

- Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters”) for some blocks.
- Simulation range checking (see “Signal Ranges” and “Enable Simulation Range Checking”).
- Automatic scaling of fixed-point data types.
- Optimization of the code that you generate from the model. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. For more information, see “Optimize using the specified minimum and maximum values” (Simulink Coder).

**Note** Output maximum does not saturate or clip the actual output signal. Use the Saturation block instead.

**Programmatic Use**

**Block Parameter:** OutMax

**Type:** character vector

**Values:** '[]' | scalar

**Default:** '[]'

Output data type — Specify the output data type

Inherit: Same as input (default) | Inherit: Inherit via back propagation | double | single | int8 | int16 | int32 | uint8 | uint16 | uint32 | int64 | uint64 | fixdt(1,16,2^0,0) | <data type expression> | ...

Choose the data type for the output. The type can be inherited, specified directly, or expressed as a data type object such as Simulink.NumericType.

**Programmatic Use**

**Block Parameter:** OutDataTypeStr

**Type:** character vector

**Values:** 'Inherit: Same as input', 'Inherit: Inherit via back propagation', 'single', 'int8', 'uint8', 'int16', 'uint16', 'int32', 'uint32',...
'int64', 'uint64', fixdt(1,16,0), fixdt(1,16,2^0,0), fixdt(1,16,2^0,0).
'<data type expression>'
**Default**: 'Inherit: Same as input'

**Lock output data type setting against changes by the fixed-point tools** — Prevent fixed-point tools from overriding Output data type

*off* (default) | *on*

Select this parameter to prevent the fixed-point tools from overriding the **Output** data type you specify on the block. For more information, see “Use Lock Output Data Type Setting” (Fixed-Point Designer).

**Programmatic Use**

**Block Parameter**: LockScale

**Type**: character vector

**Values**: 'off' | 'on'

**Default**: 'off'

**Integer rounding mode** — Specify the rounding mode for fixed-point operations

*Floor* (default) | *Ceiling* | *Convergent* | *Nearest* | *Round* | *Simplest* | *Zero*

Choose one of these rounding modes.

**Ceiling**

Rounds both positive and negative numbers toward positive infinity. Equivalent to the MATLAB **ceil** function.

**Convergent**

Rounds number to the nearest representable value. If a tie occurs, rounds to the nearest even integer. Equivalent to the Fixed-Point Designer **convergent** function.

**Floor**

Rounds both positive and negative numbers toward negative infinity. Equivalent to the MATLAB **floor** function.

**Nearest**

Rounds number to the nearest representable value. If a tie occurs, rounds toward positive infinity. Equivalent to the Fixed-Point Designer **nearest** function.

**Round**

Rounds number to the nearest representable value. If a tie occurs, rounds positive numbers toward positive infinity and rounds negative numbers toward negative infinity. Equivalent to the Fixed-Point Designer **round** function.
Simplest
Automatically chooses between round toward floor and round toward zero to generate rounding code that is as efficient as possible.

Zero
Rounds number toward zero. Equivalent to the MATLAB fix function.

**Programmatic Use**
**Block Parameter:** `RndMeth`
**Type:** character vector
**Values:** 'Ceiling' | 'Convergent' | 'Floor' | 'Nearest' | 'Round' | 'Simplest' | 'Zero'
**Default:** 'Floor'

**See Also**
For more information, see “Rounding” (Fixed-Point Designer).

**Block Characteristics**

<table>
<thead>
<tr>
<th>Data Types</th>
<th>double</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Extended Capabilities**

**C/C++ Code Generation**
Generate C and C++ code using Simulink® Coder™.
HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.
HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture
This block has a single, default HDL architecture.

HDL Block Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConstrainedOutputPipeline</td>
<td>Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. For more details, see “ConstrainedOutputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td>InputPipeline</td>
<td>Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “InputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td>OutputPipeline</td>
<td>Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “OutputPipeline” (HDL Coder).</td>
</tr>
</tbody>
</table>

PLC Code Generation
Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

See Also
Backlash | Saturation Dynamic
Introduced before R2006a
Saturation Dynamic

Limit input signal to dynamic upper and lower saturation values

**Library:**
- Simulink / Discontinuities
- HDL Coder / Discontinuities

**Description**

The Saturation Dynamic block produces an output signal that is the value of the input signal bounded to the saturation values from the input ports `up` and `lo`.

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>lo ≤ Input value ≤ hi</code></td>
<td>Input value</td>
</tr>
<tr>
<td><code>Input value &lt; lo</code></td>
<td>Lower limit</td>
</tr>
<tr>
<td><code>Input value &gt; hi</code></td>
<td>Upper limit</td>
</tr>
</tbody>
</table>

**Ports**

**Input**

* `u` — Input signal
  scalar | vector

The input signal to the saturation algorithm.

* Data Types: double | single | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

* `up` — Signal that provides the upper saturation limit
  scalar | vector

Dynamic value providing the upper saturation limit. When the input is greater than `up` then the output value is bound to `up`.
Data Types: double | single | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

lo — Signal that provides the lower saturation limit
scalar | vector

Dynamic value providing the lower saturation limit. When the input is less than lo then the output value is bound to lo.

Data Types: double | single | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

Output

Output 1 — Output signal
scalar | vector

Output signal that is the value of the input signal, upper saturation limit, or lower saturation limit.

Data Types: double | single | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

Parameters

To edit the parameters for the Saturation Dynamic block, double-click the block icon.

Click the Show data type assistant button to display the Data Type Assistant, which helps you set the data type attributes. For more information, see “Specify Data Types Using Data Type Assistant”.

Output minimum — Minimum output value for range checking
[ ] (default) | scalar

Lower value of the output range that Simulink checks.

Simulink uses the minimum to perform:

- Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters”) for some blocks.
• Simulation range checking (see “Signal Ranges” and “Enable Simulation Range Checking”).
• Automatic scaling of fixed-point data types.
• Optimization of the code that you generate from the model. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. For more information, see “Optimize using the specified minimum and maximum values” (Simulink Coder).

**Note** Output minimum does not saturate or clip the actual output signal. Use the Saturation block instead.

**Programmatic Use**

**Block Parameter:** OutMin

**Type:** character vector

**Values:** '[]' | scalar

**Default:** '[]'

**Output maximum — Maximum output value for range checking**

[] (default) | scalar

Upper value of the output range that Simulink checks.

Simulink uses the maximum value to perform:

• Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters”) for some blocks.
• Simulation range checking (see “Signal Ranges” and “Enable Simulation Range Checking”).
• Automatic scaling of fixed-point data types.
• Optimization of the code that you generate from the model. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. For more information, see “Optimize using the specified minimum and maximum values” (Simulink Coder).

**Note** Output maximum does not saturate or clip the actual output signal. Use the Saturation block instead.
Programmatic Use
Block Parameter: OutMax
Type: character vector
Values: '[ ]' | scalar
Default: '[ ]'

Output data type — Specify the output data type
Inherit: Same as input (default) | Inherit: Inherit via back propagation |
double | single | int8 | int32 | uint32 | int64 | uint64 | fixdt(1,16,2^0,0) |
<data type expression> | ...

Choose the data type for the output. The type can be inherited, specified directly, or
expressed as a data type object such as Simulink.NumericType.

Programmatic Use
Block Parameter: OutDataTypeStr
Type: character vector
Values: 'Inherit: Same as input', 'Inherit: Inherit via back
propagation', 'single', 'int8', 'uint8', 'int16', 'uint16', 'int32', 'uint32',
'int64', 'uint64', 'fixdt(1,16,0)', 'fixdt(1,16,2^0,0)', 'fixdt(1,16,2^0,0).'
'<data type expression>'
Default: 'Inherit: Same as input'

Lock output data type setting against changes by the fixed-point tools — Prevent fixed-point tools from overriding Output data type
off (default) | on

Select this parameter to prevent the fixed-point tools from overriding the Output data
type you specify on the block. For more information, see “Use Lock Output Data Type Setting” (Fixed-Point Designer).

Programmatic Use
Block Parameter: LockScale
Type: character vector
Values: 'off' | 'on'
Default: 'off'

Integer rounding mode — Specify the rounding mode for fixed-point operations
Floor (default) | Ceiling | Convergent | Nearest | Round | Simplest | Zero

Choose one of these rounding modes.
Ceiling
Rounds both positive and negative numbers toward positive infinity. Equivalent to the MATLAB ceil function.

Convergent
Rounds number to the nearest representable value. If a tie occurs, rounds to the nearest even integer. Equivalent to the Fixed-Point Designer convergent function.

Floor
Rounds both positive and negative numbers toward negative infinity. Equivalent to the MATLAB floor function.

Nearest
Rounds number to the nearest representable value. If a tie occurs, rounds toward positive infinity. Equivalent to the Fixed-Point Designer nearest function.

Round
Rounds number to the nearest representable value. If a tie occurs, rounds positive numbers toward positive infinity and rounds negative numbers toward negative infinity. Equivalent to the Fixed-Point Designer round function.

Simplest
Automatically chooses between round toward floor and round toward zero to generate rounding code that is as efficient as possible.

Zero
Rounds number toward zero. Equivalent to the MATLAB fix function.

Programmatic Use
Block Parameter: RndMeth
Type: character vector
Values: 'Ceiling' | 'Convergent' | 'Floor' | 'Nearest' | 'Round' | 'Simplest' | 'Zero'
Default: 'Floor'

See Also

For more information, see “Rounding” (Fixed-Point Designer).

Saturation on integer overflow — Choose the behavior when integer overflow occurs
on (default) | boolean
<table>
<thead>
<tr>
<th>Action</th>
<th>Reasons for Taking This Action</th>
<th>What Happens for Overflows</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Select this check box.</td>
<td>Your model has possible overflow, and you want explicit saturation protection in the generated code.</td>
<td>Overflows saturate to either the minimum or maximum value that the data type can represent.</td>
<td>The maximum value that the int8 (signed, 8-bit integer) data type can represent is 127. Any block operation result greater than this maximum value causes overflow of the 8-bit integer. With the check box selected, the block output saturates at 127. Similarly, the block output saturates at a minimum output value of -128.</td>
</tr>
<tr>
<td>Do not select this check box.</td>
<td>You want to optimize efficiency of your generated code. You want to avoid overspecifying how a block handles out-of-range signals. For more information, see “Check for Signal Range Errors”.</td>
<td>Overflows wrap to the appropriate value that is representable by the data type.</td>
<td>The maximum value that the int8 (signed, 8-bit integer) data type can represent is 127. Any block operation result greater than this maximum value causes overflow of the 8-bit integer. With the check box cleared, the software interprets the overflow-causing value as int8, which can produce an unintended result. For example, a block result of 130 (binary 1000 0010) expressed as int8, is -126.</td>
</tr>
</tbody>
</table>

When you select this check box, saturation applies to every internal operation on the block, not just the output or result. Usually, the code generation process can detect when overflow is not possible. In this case, the code generator does not produce saturation code.
Programmatic Use

**Block Parameter:** SaturateOnIntegerOverflow  
**Type:** character vector  
**Value:** 'off' | 'on'  
**Default:** 'on'

## Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>double</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
</table>

| Direct Feedthrough              | no      |        |             |         |        |
| Multidimensional Signals        | no      |        |             |         |        |
| Variable-Size Signals           | no      |        |             |         |        |
| Zero-Crossing Detection         | no      |        |             |         |        |

a. This block is not recommended for use with Boolean signals.

## Extended Capabilities

### C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

### HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

**HDL Architecture**
This block has a single, default HDL architecture.
HDL Block Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConstrainedOutputPipeline</td>
<td>Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. For more details, see “ConstrainedOutputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td>InputPipeline</td>
<td>Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “InputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td>OutputPipeline</td>
<td>Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “OutputPipeline” (HDL Coder).</td>
</tr>
</tbody>
</table>

PLC Code Generation
Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

See Also
Saturation

Introduced before R2006a
Scan String

Scan input string and convert to signals per specified format
Library: Simulink / String

Description

The Scan String block scans an input string and converts it to signals per the format specified by the Format parameter. The block converts values to their decimal (base 10) representation and outputs the results as numeric or string signals. Use this block when you want to deconstruct a string, for example a sentence, into its individual components. For example, if the Format parameter is set to "%s is %f.", the block outputs two parts, a string signal and a single signal. If the input is the string "Pi is 3.14", the two outputs are "Pi" and "3.14".

Note If you specify a numeric display format that is not large enough to display all the digits of a value, the displayed values may lose precision. The result depends on your computer hardware and operating system.

The Scan String, String to Double, and String to Single blocks are identical blocks. When configured for String to Double, the block converts the input string signal to a double numerical output. When configured for String to Single, the block converts the input string signal to a single numerical output.

For code generation, configure models that contain this block for non-finite number support by selecting the Configuration Parameters > Code Generation > Interface > Support non-finite numbers check box.
Ports

Input

Port_1 — Input string
scalar
Input string, specified as a scalar.
Data Types: string

Output

d — Output data whose format matches %d format
scalar
Output data whose format matches the specified format, defined as a scalar. Total maximum number of outputs is 128.

If the block cannot match an input string to a format operator specified in Format, it returns a warning and outputs an appropriate value (0 or "") for each unmatched format operator.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus

f — Output data whose format matches %f format
scalar
Output data whose format matches the %f format, specified as a scalar. Total maximum number of outputs is 128.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus

Port_N — Output data whose format matches N format
scalar
Output data whose format matches $N$ format, specified as a scalar. Total maximum number of outputs is 128.
Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus

Parameters

**Format — Format operator for input**
"%d %f" (default) | scalar

Format operator for input, specified as a scalar. If the block cannot match the input string with the specified format, it returns 0. The return of 0 differs from the `sscanf` function return, which is an empty matrix if the function cannot match the input with the specified format.

- For the String to Double block, this parameter has a default value of %lf.
- For the String to Single block, this parameter has a default value of %f.

For more information about acceptable format operators, see the Algorithms section.

Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>double</th>
<th>integer</th>
<th>single</th>
<th>string</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Algorithms

The Scan String block uses this format specifier prototype:

```
%[width][length]specifier
```
### Numeric Fields

This table lists available conversion specifiers to convert text to numeric outputs. The block converts values to their decimal (base 10) representation.

<table>
<thead>
<tr>
<th>Output Port Data Type</th>
<th>Conversion Specifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer, signed</td>
<td>%d</td>
<td>Base 10</td>
</tr>
<tr>
<td>Integer, unsigned</td>
<td>%u</td>
<td>Base 10</td>
</tr>
<tr>
<td>Floating-point number</td>
<td>%f, %e, or %g</td>
<td>Floating-point values. Input fields can contain NaN (case sensitive). Input fields that represents floating-point numbers can include leading + or - symbols and exponential notation using e or E. The conversion specifiers %f, %e, and %g all treat input fields the same way.</td>
</tr>
</tbody>
</table>

### Character Fields

This table lists available conversion specifiers to convert text so that the output is a string.

<table>
<thead>
<tr>
<th>Character Field Type</th>
<th>Conversion Specifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>String scalar</td>
<td>%s</td>
<td>Read the text until the block encounters whitespace.</td>
</tr>
<tr>
<td></td>
<td>%c</td>
<td>Read any single character, including whitespace. To read multiple characters at a time, specify field width. For example, %10c reads 10 characters at a time.</td>
</tr>
<tr>
<td>Character Field Type</td>
<td>Conversion Specifier</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------</td>
<td>----------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>Pattern-matching</td>
<td>%[...]</td>
<td>Read only the characters in the brackets up to the first nonmatching character or whitespace. Example: %[mus] reads 'summer' as 'summ'.</td>
</tr>
<tr>
<td></td>
<td>%[^...]</td>
<td>Read any characters not in the brackets up to the first matching character or whitespace. Example: %[^m] reads 'summer' as 'su'.</td>
</tr>
</tbody>
</table>

**Optional Operators**

- **Field Width** — To specify the maximum number of digits or text characters to read at a time, insert a number after the percent character. For example, %10s reads up to 10 characters at a time, including whitespace. %4f reads up to four digits at a time, including the decimal point.

- **Literal Text to Ignore** — This block must match the specified text immediately before or after the conversion specifier.

  Example: Hell%s reads "Hello!" as "o!".

**Length Specifiers**

The Scan String block supports the h and l length subspecifiers. These specifiers can change according to the Configuration Parameters > Hardware Implementation > Number of bits settings.

<table>
<thead>
<tr>
<th>Length</th>
<th>i</th>
<th>u</th>
<th>f e g</th>
<th>s c [...]</th>
<th>^...</th>
</tr>
</thead>
<tbody>
<tr>
<td>No length specifier</td>
<td>int</td>
<td>unsigned int</td>
<td>single</td>
<td>string</td>
<td></td>
</tr>
<tr>
<td>h</td>
<td>short</td>
<td>unsigned short</td>
<td>—</td>
<td>—</td>
<td></td>
</tr>
<tr>
<td>l</td>
<td>long</td>
<td>unsigned long</td>
<td>double</td>
<td>—</td>
<td></td>
</tr>
</tbody>
</table>
Notes for Specifiers that Specify Integer Data Types (d, u)

• Target int, long, and short type sizes are controlled by settings in the Configuration Parameters > Hardware Implementation pane. For example, if the target int is 32 bits and the specifier is %u, then the expected input type will be uint32. For this example, the Scan String block requires that the output type be exactly int32. It cannot be any other data type.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also
ASCII to String | Compose String | String Compare | String Concatenate | String Constant | String Find | String Length | String to ASCII | String to Double | String to Enum | String to Single | Substring | To String | sscanf

Topics
“Display and Extract Coordinate Data”
“Simulink Strings”

Introduced in R2018a
Scope, Time Scope

Display signals generated during simulation

**Library:**
- Simulink / Commonly Used Blocks
- Simulink / Sinks
- HDL Coder / Commonly Used Blocks
- HDL Coder / Sinks

**Description**

The Simulink Scope block and DSP System Toolbox Time Scope block display time domain signals.
The two blocks have identical functionality, but different default settings. The Time Scope is optimized for discrete time processing. The Scope is optimized for general time-domain simulation. For a side-by-side comparison, see “Simulink Scope Versus DSP System Toolbox Time Scope”.

Oscilloscope features:

- Triggers — Set triggers to sync repeating signals and pause the display when events occur.
- Cursor Measurements — Measure signal values using vertical and horizontal cursors.
- Signal Statistics — Display the maximum, minimum, peak-to-peak difference, mean, median, and RMS values of a selected signal.
- Peak Finder — Find maxima, showing the x-axis values at which they occur.
- Bilevel Measurements — Measure transitions, overshoots, undershoots, and cycles.

You must have a Simscape or DSP System Toolbox license to use the Peak Finder, Bilevel Measurements, and Signal Statistics.

Scope display features:

- Simulation control — Debug models from a Scope window using Run, Step Forward, and Step Backward toolbar buttons.
- Multiple signals — Plot multiple signals on the same y-axis (display) using multiple input ports.
- Multiple y-axes (displays) — Display multiple y-axes. All the y-axes have a common time range on the x-axis.
- Modify parameters — Modify scope parameter values before and during a simulation.
- Axis autoscaling — Autoscale axes during or at the end of a simulation. Margins are drawn at the top and bottom of the axes.
- Display data after simulation — Scope data is saved during a simulation. If a scope is closed at the start of a simulation, when you open the scope after a simulation, the scope displays simulation results for attached input signals.

**Note** If you have a high sample rate or long simulation time, you may run into issues with memory or system performance because the scope saves data internally. To limit the amount of data saved for scope visualization, use the Limit data points to last property.
For information on controlling a scope programmatically, see “Control Scope Blocks Programmatically”.

Limitations

• Do not use scope blocks in a Library. If you place a scope block inside a library block with a locked link or in a locked library, Simulink displays an error when trying to open the scope window. To display internal data from a library block, add an output port to the library block, and then connect the port to a Scope block in your model.
• If you step through a model, the scope only updates when the scope block runs. This means that the time shown in the status bar may not match the model time.
• When connected to a constant signal, a scope block may plot a single point.
• The scope shows gaps in the display when the signal value is NaN.
• When you visualize multiple frame-based signals in the scope, some samples of signals with a frame size of 1 might not be displayed. To visualize these signals, move the signals with frame size of 1 to a separate scope.

Ports

Input

Port_1 — Signal or signals to visualize
scalar | vector | matrix | array | bus | nonvirtual bus

Connect the signals you want to visualize. You can have up to 96 input ports. Input signals can have these characteristics:

• **Type** — Continuous (sample-based) or discrete (sample-based and frame-based).
• **Data type** — Any data type that Simulink supports. See “Data Types Supported by Simulink”.
• **Dimension** — Scalar, one dimensional (vector), two dimensional (matrix), or multidimensional (array). Display multiple channels within one signal depending on the dimension. See “Signal Dimensions” and “Determine Output Signal Dimensions”.

1-1730
Input Limitations

- When the input is a constant signal, the scope plots a single point.
- The scope shows gaps in the display when the signal value is NaN.
- When you visualize multiple frame-based signals in the scope, some samples of signals with a frame size of 1 might not be displayed. To visualize these signals, move the signals with frame size of 1 to a separate scope.

Bus Support

You can connect nonvirtual bus and arrays of bus signals to a scope. To display the bus signals, use normal or accelerator simulation mode. The scope displays each bus element signal in the order the elements appear in the bus, from the top to the bottom. Nested bus elements are flattened.

To log nonvirtual bus signals with a scope, set the Save format parameter to Dataset. You can use any Save format to log virtual bus signals.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus

Properties

Configuration Properties

The Configuration Properties dialog box controls various properties about the scope displays. From the scope menu, select View > Configuration Properties.

Main

Open at simulation start — Specify when scope window opens
off (default for Scope) | on (default for Time Scope)

Select this check box to open the scope window when simulation starts.

Programmatic Use

See OpenAtSimulationStart.

Display the full path — Display block path on scope title bar
off (default) | on
Select this check box to display the block path in addition to the block name.

**Number of input ports — Number of input ports on scope block**
1 (default) | integer

Specify number of input ports on a Scope block, specified as an integer. The maximum number of input ports is 96.

**Programmatic Use**

See NumInputPorts.

**Layout — Number and arrangement of displays**
1-by-1 display (default) | an arrangement of \(m\)-by-\(n\) axes

Specify number and arrangement of displays. The maximum layout is 16 rows by 16 columns.

To expand the layout grid beyond 4 by 4, click within the dialog box and drag. Maximum of 16 rows by 16 columns.

If the number of displays is equal to the number of ports, signals from each port appear on separate displays. If the number of displays is less than the number of ports, signals from additional ports appear on the last display. For layouts with multiple columns and rows, ports are mapped down then across.
Programmatic Use

See LayoutDimensions.

**Sample time — Simulation interval between scope updates**

-1 (for inherited) (default) | positive real number

Specify the time interval between updates of the scope display. This property does not apply to floating scopes and scope viewers.

Programmatic Use

See SampleTime.

**Input processing — Channel or element signal processing**

Elements as channels (sample based) (default for Scope) | Columns as channels (frame based) (default for Time Scope)

- Elements as channels (sample based) - Process each element as a unique sample.
- Columns as channels (frame based) - Process signal values in a channel as a group of values from multiple time intervals. Frame-based processing is available only with discrete input signals.

Programmatic Use

See FrameBasedProcessing.

**Maximize axes — Maximize size of plots**

Off (default for Scope) | Auto (default for Time Scope) | On

- Auto - If “Title” on page 1-0 and “Y-label” on page 1-0 properties are not specified, maximize all plots.
- On - Maximize all plots. Values in Title and Y-label are hidden.
- Off - Do not maximize plots.

Programmatic Use

See MaximizeAxes.
**Time**

**Time span — Length of x-axis to display**

Auto (default) | User defined

- **Auto** — Difference between the simulation start and stop times.

  The block calculates the beginning and end times of the time range using the “Time display offset” on page 1-0 and “Time span” on page 1-0 properties. For example, if you set the **Time display offset** to 10 and the **Time span** to 20, the scope sets the time range from 10 to 30.

- **User defined** — Enter any value less than the total simulation time.

**Programmatic Use**

See TimeSpan.

**Time span overrun action — Display data beyond visible x-axis**

Wrap (default) | Scroll

Specify how to display data beyond the visible x-axis range.

You can see the effects of this option only when plotting is slow with large models or small step sizes.

- **Wrap** — Draw a full screen of data from left to right, clear the screen, and then restart drawing the data from the left.

- **Scroll** — Move data to the left as new data is drawn on the right. This mode is graphically intensive and can affect run-time performance.

**Programmatic Use**

See TimeSpanOverrunAction.

**Time units — x-axis units**

None (default for Scope) | Metric (default for Time Scope) | Seconds

- **Metric** — Display time units based on the length of “Time span” on page 1-0.

- **Seconds** — Display time in seconds.

- **None** — Do not display time units.
Programmatic Use

See TimeUnits.

Time display offset — x-axis offset
0 (default) | scalar | vector

Offset the x-axis by a specified time value, specified as a real number or vector of real numbers.

For input signals with multiple channels, you can enter a scalar or vector:

• Scalar — Offset all channels of an input signal by the same time value.
• Vector — Independently offset the channels.

Programmatic Use

See TimeDisplayOffset.

Time-axis labels — Display of x-axis labels
Bottom Displays Only (default for Scope) | All (default for Time Scope) | None

Specify how x-axis (time) labels display:

• All — Display x-axis labels on all y-axes.
• None — Do not display labels. Selecting None also clears the Show time-axis label check box.
• Bottom displays only — Display x-axis label on the bottom y-axis.

Dependencies

To enable this property, set:

• “Show time-axis label” on page 1-0 to on.
• “Maximize axes” on page 1-0 to off.

The “Active display” on page 1-0 property determines which display is affected.

Programmatic Use

See TimeAxisLabels.
**Show time-axis label — Display or hide x-axis labels**

off (default for Scope) | on (default for Time Scope)

Select this check box to show the x-axis label for the active display

**Dependencies**

To enable this property, set “Time-axis labels” on page 1-0 to All or Bottom Displays Only.

The “Active display” on page 1-0 property determines which display is affected.

**Programmatic Use**

See ShowTimeAxisLabel.

**Display**

**Active display — Selected display**

1 (default) | positive integer

Selected display. Use this property to control which display is changed when changing style properties and axes-specific properties.

Specify the desired display using a positive integer that corresponds to the column-wise placement index. For layouts with multiple columns and rows, display numbers are mapped down and then across.

**Programmatic Use**

See ActiveDisplay.

**Title — Display name**

%<SignalLabel> (default) | string

Title for a display. The default value %<SignalLabel> uses the input signal name for the title.

**Dependency**

The “Active display” on page 1-0 property determines which display is affected.

**Programmatic Use**

See Title.
Show legend — Display signal legend
off (default) | on

Toggle signal legend. The names listed in the legend are the signal names from the model. For signals with multiple channels, a channel index is appended after the signal name. Continuous signals have straight lines before their names, and discrete signals have step-shaped lines.

From the legend, you can control which signals are visible. This control is equivalent to changing the visibility in the Style properties. In the scope legend, click a signal name to hide the signal in the scope. To show the signal, click the signal name again. To show only one signal, right-click the signal name, which hides all other signals. To show all signals, press Esc.

Note The legend only shows the first 20 signals. Any additional signals cannot be controlled from the legend.

Dependency
The “Active display” on page 1-0 property determines which display is affected.

Programmatic Use
See ShowLegend.

Show grid — Show internal grid lines
on (default) | off

Select this check box to show grid lines.

Dependency
The “Active display” on page 1-0 property determines which display is affected.

Programmatic Use
See ShowGrid.

Plot signals as magnitude and phase — Split display into magnitude and phase plots
off (default) | on
• On — Display magnitude and phase plots. If the signal is real, plots the absolute value of the signal for the magnitude. The phase is 0 degrees for positive values and 180 degrees for negative values. This feature is useful for complex-valued input signals. If the input is a real-valued signal, selecting this check box returns the absolute value of the signal for the magnitude.

• Off — Display signal plot. If the signal is complex, plots the real and imaginary parts on the same y-axis.

**Dependency**

The “Active display” on page 1-0 property determines which display is affected.

**Programmatic Use**

See `PlotAsMagnitudePhase`.

**Y-limits (Minimum) — Minimum y-axis value**

-10 (default) | real scalar

Specify the minimum value of the y-axis as a real number.

**Tunable:** Yes

**Dependency**

If you select **Plot signals as magnitude and phase**, this property only applies to the magnitude plot. The y-axis limits of the phase plot are always [ -180 180 ].

The “Active display” on page 1-0 property determines which display is affected.

**Programmatic Use**

See `YLimits`.

**Y-limits (Maximum) — Maximum y-axis value**

10 (default) | real scalar

Specify the maximum value of the y-axis as a real number.

**Tunable:** Yes

**Dependency**

If you select **Plot signals as magnitude and phase**, this property only applies to the magnitude plot. The y-axis limits of the phase plot are always [ -180 180 ].
The “Active display” on page 1-0 property determines which display is affected.

**Programmatic Use**

See YLimits.

**Y-label — Y-axis label**

none (default for Scope) | Amplitude (default for Time Scope) | string

Specify the text to display on the y-axis. To display signal units, add (%<SignalUnits>) to the label. At the beginning of a simulation, Simulink replaces (%SignalUnits) with the units associated with the signals.

Example: For a velocity signal with units of m/s, enter Velocity (%<SignalUnits>).

**Dependency**

If you select **Plot signals as magnitude and phase**, this property does not apply. The y-axes are labeled **Magnitude** and **Phase**.

The “Active display” on page 1-0 property determines which display is affected.

**Programmatic Use**

See YLabel.

**Logging**

**Limit data points to last — Limit buffered data values**

off and 5000 (default) | on | positive integer

Limit data saved by the scope internally. By default all data points are saved so that you can view the scope visualization after the simulation finishes. For simulations with **Stop time** set to inf, consider selecting **Limit data points to last**.

**Note** If you do not select **Limit data points to last** and you have a high sample rate or long simulation time, you may run into issues with memory or system performance.

When you select this property, the scope saves the latest \( n \) data points, where \( n \) the specified number of data points.

- Off — Save and plot all data values.
• On — Save specified number of data values for each signal. If the signal is frame-based, the number of buffered data values is the specified number of data values multiplied by the frame size.

In some cases, selecting this parameter can have the effect of plotting signals for less than the entire time range of a simulation (for example if your sample time is small). If the scope plots a portion of your signals, consider increasing the number of data points to save.

This property limits the data values plotted in the scope and the data values saved to a MATLAB variable specified in “Variable name” on page 1-0.

**Programmatic Use**

See `DataLoggingLimitDataPoints` and `DataLoggingMaxPoints`.

**Decimation — Reduce amount of scope data to display and save**

off, 2 (default) | on | positive integer

• On — Plot and log (save) scope data every Nth data point, where N is the decimation factor entered in the text box. A value of 1 buffers all data values.

• Off — Save all scope data values.

**Dependency**

To enable this property, select “Log data to workspace” on page 1-0.

This property limits the data values plotted in the scope and the data values saved to a MATLAB variable specified in “Variable name” on page 1-0.

**Programmatic Use**

See `DataLoggingDecimateData` and `DataLoggingDecimation`.

**Log data to workspace — Save data to MATLAB workspace**

off (default) | on

Select this check box to enable logging and enable the **Variable name**, **Save format**, and **Decimation** properties. This property does not apply to floating scopes and scope viewers.

For an example of saving signals to the MATLAB Workspace using a Scope block, see “Save Simulation Data from Floating Scope”.
**Programmatic Use**

See DataLogging.

**Variable name — Name of saved data variable**

*ScopeData (default) | string*

Specify a variable name for saving scope data in the MATLAB workspace. This property does not apply to floating scopes and scope viewers.

**Dependency**

To enable this property, select “Log data to workspace” on page 1-0.

**Programmatic Use**

See DataLoggingVariableName.

**Save format — MATLAB variable format**

*Dataset (default) | Structure With Time | Structure | Array*

Select variable format for saving data to the MATLAB workspace. This property does not apply to floating scopes and scope viewers.

- **Dataset** — Save data as a dataset object. Use the **Dataset signal format** configuration parameter to select the dataset object. This format does not support variable-size data, MAT-file logging, or external mode archiving. See `Simulink.SimulationData.Dataset`.
- **Structure With Time** — Save data as a structure with associated time information.
- **Structure** — Save data as a structure.
- **Array** — Save data as an array with associated time information. This format does not support variable-size data.

**Dependency**

To enable this property, select “Log data to workspace” on page 1-0.

**Programmatic Use**

See DataLoggingSaveFormat.
Axes Scaling Properties

The Axes Scaling dialog controls the axes limits of the scope. To open the Axes Scaling properties, in the scope menu, select Tools > Axes Scaling > Axes Scaling Properties.

Axes scaling — Y-axis scaling mode
Manual (default) | Auto | After N Updates

- **Manual** — Manually scale the y-axis range with the Scale Y-axis Limits toolbar button.
- **Auto** — Scale the y-axis range during and after simulation. Selecting this option displays the “Do not allow Y-axis limits to shrink” check box. If you want the y-axis range to increase and decrease with the maximum value of a signal, set Axes scaling to Auto and clear the Do not allow Y-axis limits to shrink check box.
- **After N Updates** — Scale y-axis after the number of time steps specified in the “Number of updates” text box (10 by default). Scaling occurs only once during each run.

Programmatic Use

See AxesScaling.

Do not allow Y-axis limits to shrink — When y-axis limits can change
on (default) | off

Allow y-axis range limits to increase but not decrease during a simulation.

Dependency

To use this property, set “Axes scaling” to Auto.

Number of updates — Number of updates before scaling
10 (default) | integer

Set this property to delay auto scaling the y-axis.

Dependency

To use this property, set “Axes scaling” to After N Updates.
Programmatic Use

See AxesScalingNumUpdates.

**Scale axes limits at stop — When y-axis limits can change**

on (default) | off

- On — Scale axes when simulation stops.
- Off — Scale axes continually.

**Dependency**

To use this property, set “Axes scaling Axes scaling” on page 1-0 to Auto.

**Y-axis Data range (%) — Percent of y-axis to use for plotting**

80 (default) | integer between [1, 100]

Specify the percentage of the y-axis range used for plotting data. If you set this property to 100, the plotted data uses the entire y-axis range.

**Y-axis Align — Alignment along y-axis**

Center (default) | Top | Bottom

Specify where to align plotted data along the y-axis data range when Y-axis Data range is set to less than 100 percent.

- Top — Align signals with the maximum values of the y-axis range.
- Center — Center signals between the minimum and maximum values.
- Bottom — Align signals with the minimum values of the y-axis range.

**Autoscale X-axis limits — Scale x-axis range limits**

off (default) | on

Scale x-axis range to fit all signal values. If Axes scaling is set to Auto, the data currently within the axes is scaled, not the entire signal in the data buffer.

**X-axis Data range (%) — Percent of x-axis to use for plotting**

100 (default) | integer in the range [1, 100]

Specify the percentage of the x-axis range to plot data on. For example, if you set this property to 100, plotted data uses the entire x-axis range.
**X-axis Align — Alignment along x-axis**
Center (default) | Top | Bottom

Specify where to align plotted data along the x-axis data range when **X-axis Data range** is set to less than 100 percent.

- **Top** — Align signals with the maximum values of the x-axis range.
- **Center** — Center signals between the minimum and maximum values.
- **Bottom** — Align signals with the minimum values of the x-axis range.

**Style Properties**

To open the Style dialog box, from the scope menu, select **View > Style**.

**Figure color — Background color for window**
black (default) | color

Background color for the scope.

**Plot type — How to plot signal**
Auto (default for Scope) | Line (default for Time Scope) | Stairs | Stem

When you select **Auto**, the plot type is a line graph for continuous signals, a stair-step graph for discrete signals, and a stem graph for Simulink message signals.

**Axes colors — Background and axes color for individual displays**
black (default) | color

Select the background color for axes (displays) with the first color palette. Select the grid and label color with the second color palette.

**Preserve colors for copy to clipboard — Copy scope without changing colors**
off (default) | on

Specify whether to use the displayed color of the scope when copying.

When you select **File > Copy to Clipboard**, the software changes the color of the scope to be printer friendly (white background, visible lines). If you want to copy and paste the scope with the colors displayed, select this check box.
Properties for line — Line to change
Channel 1 (default)

Select active line for setting line style properties.

Visible — Line visibility
on (default) | off

Show or hide a signal on the plot.

Dependency
The values of “Active display” on page 1-0 and “Properties for line” on page 1-0 determine which line is affected.

Line — Line style
solid line (default style) | 0.75 (default width) | yellow (default color)

Select line style, width, and color.

Dependency
The values of “Active display” on page 1-0 and “Properties for line” on page 1-0 determine which line is affected.

Marker — Data point marker style
None (default) | marker shape

Select marker shape.

Dependency
The values of “Active display” on page 1-0 and “Properties for line” on page 1-0 determine which line is affected.

Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>bus_a</th>
<th>double</th>
<th>enumerated</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
</table>

1-1745
<table>
<thead>
<tr>
<th>Feature</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>yes</td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>yes</td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
</tr>
</tbody>
</table>

a. Virtual bus not supported. Nonvirtual bus supported only in normal and accelerator mode simulation. Data logging for nonvirtual bus supported only in the dataset format.

Extended Capabilities

**C/C++ Code Generation**
Generate C and C++ code using Simulink® Coder™.

This block can be used for simulation visibility in systems that generate code, but is not included in the generated code.

**HDL Code Generation**
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

This block can be used for simulation visibility in subsystems that generate HDL code, but is not included in the hardware implementation.

**PLC Code Generation**
Generate Structured Text code using Simulink® PLC Coder™.

This block can be used for simulation visibility in systems that generate code, but is not included in the generated code.
Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

This block accepts fixed-point input, but converts it to double for display.

See Also
Floating Scope | Scope Viewer

Topics
“Scope Blocks and Scope Viewer Overview”
“Step Through a Simulation”
“Common Scope Block Tasks”
“Control Scope Blocks Programmatically”
Scope Block with Apple iOS Devices (Simulink Support Package for Apple iOS Devices)
“Scope Block with Android Devices” (Simulink Support Package for Android Devices)
“Simulink Scope Block and Mobile Apps” (Simulink Support Package for Android Devices)

Introduced in R2015b
Second-Order Integrator

Integrate input signal twice

**Library:** Simulink / Continuous

---

**Description**

The Second-Order Integrator block and the Second-Order Integrator Limited block solve the second-order initial value problem:

\[
\frac{d^2x}{dt^2} = u,
\]

\[
\frac{dx}{dt}\bigg|_{t=0} = dx_0,
\]

\[
x|_{t=0} = x_0,
\]

where \( u \) is the input to the system. The block is therefore a dynamic system with two continuous states: \( x \) and \( dx/dt \).

**Note** These two states have a mathematical relationship, namely, that \( dx/dt \) is the derivative of \( x \). To satisfy this relationship throughout the simulation, Simulink places various constraints on the block parameters and behavior.

The Second-Order Integrator Limited block is identical to the Second-Order Integrator block with the exception that it defaults to limiting the states based on the specified upper and lower limits. For more information, see “Limiting the States” on page 1-1750.

Simulink software can use several different numerical integration methods to compute the outputs of the block. Each has advantages for specific applications. Use the **Solver** pane of the Configuration Parameters dialog box to select the technique best suited to your application. (For more information, see “Solver Selection Criteria”.) The selected
solver computes the states of the Second-Order Integrator block at the current time step using the current input value.

Use the block parameter dialog box to:

- Specify whether the source of each state initial condition is internal or external
- Specify a value for the state initial conditions
- Define upper and lower limits on either or both states
- Specify absolute tolerances for each state
- Specify names for both states
- Choose an external reset condition
- Enable zero-crossing detection
- Reinitialize dx/dt when x reaches saturation
- Specify that Simulink disregard the state limits and external reset for linearization operations

**Defining Initial Conditions**

You can define the initial conditions of each state individually as a parameter on the block dialog box or input one or both of them from an external signal.

- To define the initial conditions of state \( x \) as a block parameter, use the **Initial condition source** \( x \) drop-down menu to select **internal** and enter the value in the **Initial condition** \( x \) field.
- To provide the initial conditions from an external source for state \( x \), specify the **Initial condition source** \( x \) parameter as **external**. An additional input port appears on the block.
- To define the initial conditions of state \( dx/dt \) as a block parameter, use the **Initial condition source** \( dx/dt \) drop-down menu to select **internal** and enter the value in the **Initial condition** \( dx/dt \) field.
- To provide the initial conditions from an external source for state \( dx/dt \), specify **Initial condition source** \( dx/dt \) as **external**. An additional input port appears on the block.

If you choose to use an external source for both state initial conditions, your block appears as follows.
Note

- Simulink does not allow initial condition values of `inf` or `NaN`.
- If you limit state \( x \) or state \( dx/dt \) by specifying saturation limits (see “Limiting the States” on page 1-1750) and one or more initial conditions are outside the corresponding limits, then the respective states are initialized to the closest valid value and a set of consistent initial conditions is calculated.

Limiting the States

When modeling a second-order system, you may need to limit the block states. For example, the motion of a piston within a cylinder is governed by Newton's Second Law and has constraints on the piston position \( (x) \). With the Second-Order Integrator block, you can limit the states \( x \) and \( dx/dt \) independent of each other. You can even change the limits during simulation; however, you cannot change whether or not the states are limited. An important rule to follow is that an upper limit must be strictly greater than its corresponding lower limit.

The block appearance changes when you limit one or both states. With both states limited, the block appears as follows.
For each state, you can use the block parameter dialog box to set appropriate saturation limits.

**Limiting x Only**

If you use the Second-Order Integrator Limited block, both states are limited by default. But you can also manually limit state \( x \) on the Second-Order Integrator block by selecting **Limit x** and entering the limits in the appropriate parameter fields.
The block then determines the values of the states as follows:
• When $x$ is less than or equal to its lower limit, the value of $x$ is held at its lower limit and $\frac{dx}{dt}$ is set to zero.

• When $x$ is in between its lower and upper limits, both states follow the trajectory given by the second-order ODE.

• When $x$ is greater than or equal to its upper limit, the value of $x$ is held at its upper limit and $\frac{dx}{dt}$ is set to zero.

You can choose to reinitialize $\frac{dx}{dt}$ to a new value at the time when $x$ reaches saturation. See “Reinitializing $\frac{dx}{dt}$ When $x$ Reaches Saturation” on page 1-1757.

**Limiting $\frac{dx}{dt}$ Only**

As with state $x$, state $\frac{dx}{dt}$ is set as limited by default on the $\frac{dx}{dt}$ pane of the Second-Order Integrator Limited block dialog box. You can manually set this parameter, Limit $\frac{dx}{dt}$, on the Second-Order Integrator block. In either case, you must enter the appropriate limits for $\frac{dx}{dt}$. 
If you limit only the state $dx/dt$, then the block determines the values of $dx/dt$ as follows:
• When $dx/dt$ is less than or equal to its lower limit, the value of $dx/dt$ is held at its lower limit.
• When $dx/dt$ is in between its lower and upper limits, both states follow the trajectory given by the second-order ODE.
• When $dx/dt$ is greater than or equal to its upper limit, the value of $dx/dt$ is held at its upper limit.

When state $dx/dt$ is held at it upper or lower limit, the value of $x$ is governed by the first-order initial value problem:

$$\frac{dx}{dt} = L,$$
$$x(t_L) = x_L,$$

where $L$ is the $dx/dt$ limit (upper or lower), $t_L$ is the time when $dx/dt$ reaches this limit, and $x_L$ is the value of state $x$ at that time.

**Limiting Both States**

When you limit both states, Simulink maintains mathematical consistency of the states by limiting the allowable values of the upper and lower limits for $dx/dt$. Such limitations are necessary to satisfy the following constraints:

• When $x$ is at its saturation limits, the value of $dx/dt$ must be zero.
• In order for $x$ to leave the upper limit, the value of $dx/dt$ must be strictly negative.
• In order for $x$ to leave its lower limit, the value of $dx/dt$ must be strictly positive.

For such cases, the upper limit of $dx/dt$ must be strictly positive and the lower limit of $dx/dt$ must be strictly negative.

When both states are limited, the block determines the states as follows:

• Whenever $x$ reaches its limits, the resulting behavior is the same as that described in “Limiting $x$ only”.
• Whenever $dx/dt$ reaches one of its limits, the resulting behavior is the same as that described in “Limiting $dx/dt$ only” — including the computation of $x$ using a first-order ODE when $dx/dt$ is held at one of its limits. In such cases, when $x$ reaches one of its limits, it is held at that limit and $dx/dt$ is set to zero.
Whenever both reach their respective limits simultaneously, the state $x$ behavior overrides $dx/dt$ behavior to maintain consistency of the states.

When you limit both states, you can choose to reinitialize $dx/dt$ at the time when state $x$ reaches saturation. If the reinitialized value is outside specified limits on $dx/dt$, then $dx/dt$ is reinitialized to the closest valid value and a consistent set of initial conditions is calculated. See “Reinitializing dx/dt When x Reaches Saturation” on page 1-1757

**Resetting the State**

The block can reset its states to the specified initial conditions based on an external signal. To cause the block to reset its states, select one of the **External reset** choices on the Attributes pane. A trigger port appears on the block below its input port and indicates the trigger type.

- Select **rising** to reset the states when the reset signal rises from zero to a positive value, from a negative to a positive value, or a negative value to zero.
- Select **falling** to reset the states when the reset signal falls from a positive value to zero, from a positive to a negative value, or from zero to negative.
- Select **either** to reset the states when the reset signal changes from zero to a nonzero value or changes sign.

The reset port has direct feedthrough. If the block output feeds back into this port, either directly or through a series of blocks with direct feedthrough, an algebraic loop results (see “Algebraic Loop Concepts”).
Enabling Zero-Crossing Detection

This parameter controls whether zero-crossing detection is enabled for this block. By default, the Enable zero-crossing detection parameter is selected on the Attributes pane. However, this parameter is only in affect if the Zero-crossing control, on the Solver pane of the Configuration Parameters dialog box, is set to Use local settings. For more information, see “Zero-Crossing Detection”.

Reinitializing $dx/dt$ When $x$ Reaches Saturation

For certain modeling applications, $dx/dt$ must be reinitialized when state $x$ reaches its limits in order to pull $x$ out of saturation immediately. You can achieve this by selecting Reinitialize $dx/dt$ when $x$ reaches saturation on the Attributes pane.

If this option is on, then at the instant when $x$ reaches saturation, Simulink checks whether the current value of the $dx/dt$ initial condition (parameter or signal) allows the state $x$ to leave saturation immediately. If so, Simulink reinitializes state $dx/dt$ with the value of the initial condition (parameter or signal) at that instant. If not, Simulink ignores this parameter at the current instant and sets $dx/dt$ to zero to make the block states consistent.

This parameter only applies at the time when $x$ actually reaches saturation limit. It does not apply at any future time when $x$ is being held at saturation.

Refer to the sections on limiting the states for more information. For an example, see “Simulation of a Bouncing Ball” on page 15-7.

Disregarding State Limits and External Reset for Linearization

For cases where you simplify your model by linearizing it, you can have Simulink disregard the limits of the states and the external reset by selecting Ignore state limits and the reset for linearization.

Specifying the Absolute Tolerance for the Block Outputs

By default Simulink software uses the absolute tolerance value specified in the Configuration Parameters dialog box (see “Error Tolerances for Variable-Step Solvers”) to compute the output of the integrator blocks. If this value does not provide sufficient error control, specify a more appropriate value for state $x$ in the Absolute tolerance $x$ field.
and for state $dx/dt$ in the **Absolute tolerance** $dx/dt$ field of the parameter dialog box. Simulink uses the values that you specify to compute the state values of the block.

### Specifying the Display of the Output Ports

You can control whether to display the $x$ or the $dx/dt$ output port using the **ShowOutput** parameter. You can display one output port or both; however, you must select at least one.

### Specifying the State Names

You can specify the name of $x$ states and $dx/dt$ states using the **StateNameX** and **StateNameDXDT** parameters. However, you must specify names for both or neither; you cannot specify names for just $x$ or just $dx/dt$. Both state names must have identical type and length. Furthermore, the number of names must evenly divide the number of states.

### Selecting All Options

When you select all options, the block icon looks like this.

![Integrator, Second-Order Block Icon](image)

### Ports

#### Input

 greentext

$u$ — Input signal $u$

- scalar
- vector
- matrix
Input signal \( u \) to the integrator system, specified as a scalar, vector, or matrix.

Data Types: double

\( x_0 \) — Initial condition \( x_0 \)

scalar | vector | matrix

External signal specifying the initial condition \( x_0 \) to the integrator system. You can specify the initial condition as a scalar, vector, or matrix.

Dependencies

To enable this input port, set the Initial condition source \( x \) parameter to external.

Data Types: double

\( dx_0 \) — Initial condition \( dx_0 \)

scalar | vector | matrix

External signal specifying the initial condition \( dx_0 \) to the integrator system. You can specify the initial condition \( dx_0 \) as a scalar, vector, or matrix.

Dependencies

To enable this input port, set the Initial condition source \( dx/dt \) parameter to external.

Data Types: double

Output

\( x \) — Output signal \( x \)

scalar | vector | matrix

\( x \) state output signal, provided as a scalar, vector, or matrix.

Data Types: double

\( dx \) — Output signal \( dx \)

scalar | vector | matrix

\( dx \) state output signal, specified as a scalar, vector, or matrix.

Data Types: double
**Parameters**

**x**

*Initial condition source x — Source of initial condition for state x*

- **internal** (default) | **external**

 Specify the source of the initial conditions for state x.

- **internal** — Get the initial conditions of state x from the **Initial condition x** parameter.
- **external** — Get the initial conditions of state x from an external block connected to the **X0** input port.

**Limitations**

Simulink software does not allow the initial condition of this block to be inf or NaN.

**Dependencies**

Selecting **internal** enables the **Initial condition x** parameter and removes the **X0** input port.

Selecting **external** disables the **Initial condition x** parameter and enables the **X0** input port.

**Programmatic Use**

**Block Parameter:** **ICSourceX**

- **Type:** character vector, string
- **Values:** 'internal' | 'external'
- **Default:** 'internal'

*Initial condition x — Initial condition of state x*

- **0.0** (default) | **scalar** | **vector** | **matrix**

Specify the initial condition of state x.

**Limitations**

Simulink software does not allow the initial condition of this block to be inf or NaN.
Dependencies

To enable this parameter, set Initial condition source \( x \) to internal.

Setting Initial condition source \( x \) to external disables this parameter and enables the \( X_0 \) input port.

Programmatic Use
Block Parameter: ICX
Type: character vector, string
Values: scalar | vector | matrix
Default: '0.0'

Limit \( x \) — Limit values of state \( x \)
off | on

Limit state \( x \) of the block to a value between the Lower limit \( x \) and Upper limit \( x \) parameters. The default value of the Second-Order Integrator block is off. The default value of the Second-Order Integrator Limited is on.

- To limit state \( x \) to a value between the Lower limit \( x \) and Upper limit \( x \) parameters, select this check box.
- To remove range limitations on state \( x \), clear this check box.

Dependencies

Selecting this check box enables the Upper limit \( x \) and Lower limit \( x \) parameters.

Programmatic Use
Block Parameter: LimitX
Type: character vector, string
Values: 'off' | 'on'
Default: 'off' (Second-Order Integrator) | 'on' (Second-Order Integrator Limited)

Upper limit \( x \) — Upper limit of state \( x \)
1 | inf | scalar | vector | matrix

Specify the upper limit of state \( x \). The default value for the Second-Order Integrator block is inf. The default value for the Second-Order Integrator Limited block is 1.

Tips

The upper saturation limit for state \( x \) must be strictly greater than the lower saturation limit.
Dependencies

To enable this parameter, select the Limit x check box.

Programmatic Use
Block Parameter: UpperLimitX
Type: character vector, string
Values: '1' | 'inf' | scalar | vector | matrix
Default: '1' (Second-Order Integrator Limited) | 'inf' (Second-Order Integrator)

Lower limit x — Lower limit of state x
θ (default) | -inf | scalar | vector | matrix

Specify the lower limit of state x. The default value for the Second-Order Integrator block is -inf. The default value for the Second-Order Integrator Limited block is θ.

Tip
The lower saturation limit for state x must be strictly less than the upper saturation limit.

Dependencies
To enable this parameter, select the Limit x check box.

Programmatic Use
Block Parameter: LowerLimitX
Type: character vector, string
Values: 'θ' | '-inf' | scalar | vector | matrix
Default: 'θ' (Second-Order Integrator Limited) | '-inf' (Second-Order Integrator)

Wrap x — Enable wrapping of x
off (default) | on

Enable wrapping of x between the Wrapped upper value x and Wrapped lower value x parameters. Enabling wrapping of x eliminates the need for zero-crossing detection, reduces solver resets, improves solver performance and accuracy, and increases simulation time span when modeling rotary and cyclic state trajectories.

If you specify Wrapped upper value x as inf and Wrapped lower value x as -inf, wrapping will never occur.

Dependencies
Selecting this check box enables Wrapped upper value x and Wrapped lower value x.
Programmatic Use
Block Parameter: WrapX
Type: character vector, string
Values: 'off' | 'on'
Default: 'off'

Wrapped upper value x — Upper value for wrapping x
	pi (default) | scalar | vector | matrix

Specify the upper value for wrapping x.

Dependencies
To enable this parameter, select the Wrap x check box.

Programmatic Use
Block Parameter: WrappedUpperValueX
Type: character vector, string
Values: scalar | vector | matrix
Default: 'pi'

Wrapped lower value x — Lower value for wrapping x

-pi (default) | scalar | vector | matrix

Specify the lower value for wrapping x.

Dependencies
To enable this parameter, select the Wrap x check box.

Programmatic Use
Block Parameter: WrappedLowerValueX
Type: character vector, string
Values: scalar | vector | matrix
Default: '-pi'

Absolute tolerance x — Absolute tolerance for computing state x
auto (default) | -1 | scalar | vector

Specify the absolute tolerance for computing state x.

- You can enter auto, -1, a positive real scalar or vector.
- If you enter auto or -1, Simulink uses the absolute tolerance value in the
  Configuration Parameters dialog box (see “Solver Pane”) to compute state x.
• If you enter a real scalar value, that value overrides the absolute tolerance in the Configuration Parameters dialog box and is used for computing all x states.

• If you enter a real vector, the dimension of that vector must match the dimension of state x. These values override the absolute tolerance in the Configuration Parameters dialog box.

Programmatic Use

Block Parameter: AbsoluteToleranceX
Type: character vector; string
Values: 'auto' | '-' | any positive real scalar or vector
Default: 'auto'

State name x — Name for state x
' ' (default) | character vector | string

Assign a unique name to state x.

Tips

• To assign a name to a single state, enter the name between quotes, for example, position'.

• To assign names to multiple x states, enter a comma-delimited list surrounded by braces, for example, {'a', 'b', 'c'}. Each name must be unique.

• The state names apply only to the selected block.

• If you specify a state name for x, you must also specify a state name for dx/dt.

• State names for x and dx/dt must have identical types and lengths.

• The number of states must be evenly divided by the number of state names. You can specify fewer names than x states, but you cannot specify more names than x states. For example, you can specify two names in a system with four states. The first name applies to the first two states and the second name to the last two states. However, you must be consistent and apply the same scheme to the state names for dx/dt.

• To assign state names with a variable in the MATLAB workspace, enter the variable without quotes. A variable can be a character vector, string or a cell array.

Programmatic Use

Block Parameter: StateNameX
Type: character vector; string
Values: ' ' | user-defined
Default: ''
**dx/dt**

**Initial condition source dx/dt — Source of initial condition for state dx/dt**
*internal* (default) | *external*

Specify the source of initial conditions for state \( dx/dt \) as *internal* or *external*.

**Limitations**

Simulink software does not allow the initial condition of this block to be *inf* or *NaN*.

**Dependencies**

- Selecting *internal* enables the **Initial condition dx/dt** parameter and removes the \( dx_0 \) input port.
- Selecting *external* disables the **Initial condition dx/dt** parameter and enables the \( dx_0 \) input port.

**Programmatic Use**

*Block Parameter: ICSourceDXDT*
*Type: character vector*
*Values: 'internal' | 'external'*
*Default: 'internal'*

**Initial condition dx/dt — Initial condition of state dx/dt**
*0.0 (default) | scalar | vector | matrix*

Specify the initial condition of state \( dx/dt \).

**Limitations**

Simulink software does not allow the initial condition of this block to be *inf* or *NaN*.

**Dependencies**

- Setting **Initial condition source dx/dt** to *internal* enables this parameter.
- Setting **Initial condition source dx/dt** to *external* disables this parameter.

**Programmatic Use**

*Block Parameter: ICDXDT*
*Type: character vector*
*Values: scalar | vector | matrix*
*Default: '0.0'
**Limit dx/dt — Limit values of state dx/dt**

off | on

Limit the $dx/dt$ state of the block to a value between the **Lower limit dx/dt** and **Upper limit dx/dt** parameters. The default value of the Second-Order Integrator block is off. The default value of the Second-Order Integrator Limited is on.

**Tip**

If you set saturation limits for $x$, then the interval defined by the **Upper limit dx/dt** and **Lower limit dx/dt** must contain zero.

**Dependencies**

Selecting this check box enables the **Upper limit dx/dt** and **Lower limit dx/dt** parameters.

**Programmatic Use**

**Parameter:** LimitDXDT  
**Type:** character vector  
**Values:** 'off' | 'on'  
**Default:** 'off' (Second-Order Integrator) | 'on' (Second-Order Integrator Limited)

**Upper limit dx/dt — Upper limit of state dx/dt**

$\infty$ (default) | scalar | vector | matrix

Specify the upper limit for state $dx/dt$.

**Dependencies**

If you limit $x$, then this parameter must have a strictly positive value.

To enable this parameter, select the **Limit dx/dt** check box.

**Programmatic Use**

**Block Parameter:** UpperLimitDXDT  
**Type:** character vector  
**Values:** scalar | vector | matrix  
**Default:** '$\infty$'

**Lower limit dx/dt — Lower limit of state dx/dt**

$-\infty$ (default) | scalar | vector | matrix

Specify the lower limit for state $dx/dt$. 
Dependencies

If you limit x, then this parameter must have a strictly negative value.

To enable this parameter, select the Limit dx/dt check box.

Programmatic Use

Block Parameter: LowerLimitDXDT
Type: character vector
Values: scalar | vector | matrix
Default: '-inf'

Absolute tolerance dx/dt — Absolute tolerance for computing state dx/dt
auto (default) | -1 | scalar | vector

Specify the absolute tolerance for computing state dx/dt.

- You can enter auto, –1, a positive real scalar or vector.
- If you enter auto or –1, then Simulink uses the absolute tolerance value in the Configuration Parameters dialog box (see “Solver Pane”) to compute the dx/dt output of the block.
- If you enter a numeric value, that value overrides the absolute tolerance in the Configuration Parameters dialog box.

Programmatic Use

Block Parameter: AbsoluteToleranceDXDT
Type: character vector, string, scalar, or vector
Values: 'auto' | '-1' | any positive real scalar or vector
Default: 'auto'

State name dx/dt — Name for state dx/dt
' ' (default) | character vector | string

Assign a unique name to state dx/dt.

Tips

- To assign a name to a single state, enter the name between quotes, for example, 'velocity'.
- To assign names to multiple dx/dt states, enter a comma-delimited list surrounded by braces, for example, {'a', 'b', 'c'}. Each name must be unique.
• The state names apply only to the selected block.
• If you specify a state name for \( dx/dt \), you must also specify a state name for \( x \).
• State names for \( x \) and \( dx/dt \) must have identical types and lengths.
• The number of states must be evenly divided by the number of state names. You can specify fewer names than \( dx/dt \) states, but you cannot specify more names than \( dx/dt \) states. For example, you can specify two names in a system with four states. The first name applies to the first two states and the second name to the last two states. However, you must be consistent and apply the same scheme to the state names for \( x \).
• To assign state names with a variable in the MATLAB workspace, enter the variable without quotes. A variable can be a character vector, string, or a cell array.

**Programmatic Use**

**Block Parameter:** StateNameDXDT  
**Type:** character vector, string  
**Values:** ' ' | user-defined  
**Default:** ' '

**Attributes**

**External reset — Reset states to their initial conditions**  
none (default) | rising | falling | either

Reset the states to their initial conditions when a trigger event occurs in the reset signal.

- none — Do not reset the state to initial conditions.
- rising — Reset the state when the reset signal rises from a zero to a positive value or from a negative to a positive value.
- falling — Reset the state when the reset signal falls from a positive value to zero or from a positive to a negative value.
- either — Reset the state when the reset signal changes from zero to a nonzero value or changes sign.

**Programmatic Use**

**Block Parameter:** ExternalReset  
**Type:** character vector, string  
**Values:** 'none' | 'rising' | 'falling' | 'either'  
**Default:** 'none'
Enable zero-crossing detection — Enable zero-crossing detection
on (default) | off

Select to enable zero-crossing detection. For more information, see “Zero-Crossing Detection”.

**Programmatic Use**
**Parameter:** ZeroCross  
**Type:** character vector, string  
**Values:** 'on' | 'off'  
**Default:** 'on'

Reinitialize dx/dt when x reaches saturation — Reset dx/dt when x reaches saturation
off (default) | on

At the instant when state x reaches saturation, reset dx/dt to its current initial conditions.

**Tip**

The dx/dt initial condition must have a value that enables x to leave saturation immediately. Otherwise, Simulink ignores the initial conditions for dx/dt to preserve mathematical consistency of block states.

**Programmatic Use**
**Block Parameter:** ReinitDXDTwhenXreachesSaturation  
**Type:** character vector, string  
**Values:** 'off' | 'on'  
**Default:** 'off'

Ignore state limits and the reset for linearization — Ignore state limits and external reset for linearization
off (default) | on

For linearization purposes, have Simulink ignore the specified state limits and the external reset.

**Programmatic Use**
**Block Parameter:** IgnoreStateLimitsAndResetForLinearization  
**Type:** character vector, string  
**Values:** 'off' | 'on'  
**Default:** 'off'
Show output — Output ports to display
both (default) | x | dx/dt

Specify the output ports on the block.

- both — Show both x and dx/dt output ports.
- x — Show only the x output port.
- dx/dt — Show only the dx/dt output port.

Programmatic Use
Block Parameter: ShowOutput
Type: character vector, string
Values: 'both' | 'x' | 'dxdt'
Default: 'both'

Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>double</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>yes</td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>no</td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>yes</td>
</tr>
</tbody>
</table>

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Not recommended for production-quality code. Relates to resource limits and restrictions on speed and memory often found in embedded systems. The code generated can contain
dynamic allocation and freeing of memory, recursion, additional memory overhead, and widely-varying execution times. While the code is functionally valid and generally acceptable in resource-rich environments, smaller embedded targets often cannot support such code.

In general, consider using the Simulink Model Discretizer to map continuous blocks into discrete equivalents that support production code generation. To start the Model Discretizer, select Analysis > Control Design > Model Discretizer. One exception is the Second-Order Integrator block because, for this block, the Model Discretizer produces an approximate discretization.

**See Also**
Integrator | Integrator Limited | Second-Order Integrator Limited

**Topics**
“Zero-Crossing Detection”
“Error Tolerances for Variable-Step Solvers”
“Algebraic Loop Concepts”

**Introduced in R2010a**
Second-Order Integrator Limited

Integrate input signal twice
Library: Simulink / Continuous

Description

The Second-Order Integrator block and the Second-Order Integrator Limited block solve the second-order initial value problem:

\[ \frac{d^2x}{dt^2} = u, \]
\[ \left. \frac{dx}{dt} \right|_{t=0} = d\xi, \]
\[ x|_{t=0} = x_0, \]

where \( u \) is the input to the system. The block is therefore a dynamic system with two continuous states: \( x \) and \( dx/dt \).

Note These two states have a mathematical relationship, namely, that \( dx/dt \) is the derivative of \( x \). To satisfy this relationship throughout the simulation, Simulink places various constraints on the block parameters and behavior.

The Second-Order Integrator Limited block is identical to the Second-Order Integrator block with the exception that it defaults to limiting the states based on the specified upper and lower limits. For more information, see “Limiting the States” on page 1-1774.

Simulink software can use several different numerical integration methods to compute the outputs of the block. Each has advantages for specific applications. Use the Solver pane of the Configuration Parameters dialog box to select the technique best suited to your application. (For more information, see “Solver Selection Criteria”.) The selected
solver computes the states of the Second-Order Integrator block at the current time step using the current input value.

Use the block parameter dialog box to:

- Specify whether the source of each state initial condition is internal or external
- Specify a value for the state initial conditions
- Define upper and lower limits on either or both states
- Specify absolute tolerances for each state
- Specify names for both states
- Choose an external reset condition
- Enable zero-crossing detection
- Reinitialize dx/dt when x reaches saturation
- Specify that Simulink disregard the state limits and external reset for linearization operations

**Defining Initial Conditions**

You can define the initial conditions of each state individually as a parameter on the block dialog box or input one or both of them from an external signal.

- To define the initial conditions of state x as a block parameter, use the **Initial condition source x** drop-down menu to select **internal** and enter the value in the **Initial condition x** field.
- To provide the initial conditions from an external source for state x, specify the **Initial condition source x** parameter as **external**. An additional input port appears on the block.
- To define the initial conditions of state dx/dt as a block parameter, use the **Initial condition source dx/dt** drop-down menu to select **internal** and enter the value in the **Initial condition dx/dt** field.
- To provide the initial conditions from an external source for state dx/dt, specify **Initial condition source dx/dt** as **external**. An additional input port appears on the block.

If you choose to use an external source for both state initial conditions, your block appears as follows.
Note

- Simulink does not allow initial condition values of \( \text{inf} \) or \( \text{NaN} \).
- If you limit state \( x \) or state \( dx/dt \) by specifying saturation limits (see “Limiting the States” on page 1-1774) and one or more initial conditions are outside the corresponding limits, then the respective states are initialized to the closest valid value and a set of consistent initial conditions is calculated.

Limiting the States

When modeling a second-order system, you may need to limit the block states. For example, the motion of a piston within a cylinder is governed by Newton's Second Law and has constraints on the piston position \( (x) \). With the Second-Order Integrator block, you can limit the states \( x \) and \( dx/dt \) independent of each other. You can even change the limits during simulation; however, you cannot change whether or not the states are limited. An important rule to follow is that an upper limit must be strictly greater than its corresponding lower limit.

The block appearance changes when you limit one or both states. With both states limited, the block appears as follows.
For each state, you can use the block parameter dialog box to set appropriate saturation limits.

**Limiting x Only**

If you use the Second-Order Integrator Limited block, both states are limited by default. But you can also manually limit state $x$ on the Second-Order Integrator block by selecting **Limit x** and entering the limits in the appropriate parameter fields.
The block then determines the values of the states as follows:
• When \( x \) is less than or equal to its lower limit, the value of \( x \) is held at its lower limit and \( dx/dt \) is set to zero.

• When \( x \) is in between its lower and upper limits, both states follow the trajectory given by the second-order ODE.

• When \( x \) is greater than or equal to its upper limit, the value of \( x \) is held at its upper limit and \( dx/dt \) is set to zero.

You can choose to reinitialize \( dx/dt \) to a new value at the time when \( x \) reaches saturation. See “Reinitializing \( dx/dt \) When \( x \) Reaches Saturation” on page 1-1781.

**Limiting \( dx/dt \) Only**

As with state \( x \), state \( dx/dt \) is set as limited by default on the \( dx/dt \) pane of the Second-Order Integrator Limited block dialog box. You can manually set this parameter, **Limit \( dx/dt \)**, on the Second-Order Integrator block. In either case, you must enter the appropriate limits for \( dx/dt \).
If you limit only the state $dx/dt$, then the block determines the values of $dx/dt$ as follows:
When \( \frac{dx}{dt} \) is less than or equal to its lower limit, the value of \( \frac{dx}{dt} \) is held at its lower limit.

When \( \frac{dx}{dt} \) is in between its lower and upper limits, both states follow the trajectory given by the second-order ODE.

When \( \frac{dx}{dt} \) is greater than or equal to its upper limit, the value of \( \frac{dx}{dt} \) is held at its upper limit.

When state \( \frac{dx}{dt} \) is held at its upper or lower limit, the value of \( x \) is governed by the first-order initial value problem:

\[
\frac{dx}{dt} = L, \\
x(t_L) = x_L,
\]

where \( L \) is the \( \frac{dx}{dt} \) limit (upper or lower), \( t_L \) is the time when \( \frac{dx}{dt} \) reaches this limit, and \( x_L \) is the value of state \( x \) at that time.

### Limiting Both States

When you limit both states, Simulink maintains mathematical consistency of the states by limiting the allowable values of the upper and lower limits for \( \frac{dx}{dt} \). Such limitations are necessary to satisfy the following constraints:

- When \( x \) is at its saturation limits, the value of \( \frac{dx}{dt} \) must be zero.
- In order for \( x \) to leave the upper limit, the value of \( \frac{dx}{dt} \) must be strictly negative.
- In order for \( x \) to leave its lower limit, the value of \( \frac{dx}{dt} \) must be strictly positive.

For such cases, the upper limit of \( \frac{dx}{dt} \) must be strictly positive and the lower limit of \( \frac{dx}{dt} \) must be strictly negative.

When both states are limited, the block determines the states as follows:

- Whenever \( x \) reaches its limits, the resulting behavior is the same as that described in “Limiting \( x \) only”.
- Whenever \( \frac{dx}{dt} \) reaches one of its limits, the resulting behavior is the same as that described in “Limiting \( \frac{dx}{dt} \) only” — including the computation of \( x \) using a first-order ODE when \( \frac{dx}{dt} \) is held at one of its limits. In such cases, when \( x \) reaches one of its limits, it is held at that limit and \( \frac{dx}{dt} \) is set to zero.
Whenever both reach their respective limits simultaneously, the state $x$ behavior overrides $dx/dt$ behavior to maintain consistency of the states.

When you limit both states, you can choose to reinitialize $dx/dt$ at the time when state $x$ reaches saturation. If the reinitialized value is outside specified limits on $dx/dt$, then $dx/dt$ is reinitialized to the closest valid value and a consistent set of initial conditions is calculated. See “Reinitializing $dx/dt$ When $x$ Reaches Saturation” on page 1-1781.

**Resetting the State**

The block can reset its states to the specified initial conditions based on an external signal. To cause the block to reset its states, select one of the **External reset** choices on the Attributes pane. A trigger port appears on the block below its input port and indicates the trigger type.

- Select **rising** to reset the states when the reset signal rises from zero to a positive value, from a negative to a positive value, or a negative value to zero.
- Select **falling** to reset the states when the reset signal falls from a positive value to zero, from a positive to a negative value, or from zero to negative.
- Select **either** to reset the states when the reset signal changes from zero to a nonzero value or changes sign.

The reset port has direct feedthrough. If the block output feeds back into this port, either directly or through a series of blocks with direct feedthrough, an algebraic loop results (see “Algebraic Loop Concepts”).
Enabling Zero-Crossing Detection

This parameter controls whether zero-crossing detection is enabled for this block. By default, the Enable zero-crossing detection parameter is selected on the Attributes pane. However, this parameter is only in affect if the Zero-crossing control, on the Solver pane of the Configuration Parameters dialog box, is set to Use local settings. For more information, see “Zero-Crossing Detection”.

Reinitializing $dx/dt$ When $x$ Reaches Saturation

For certain modeling applications, $dx/dt$ must be reinitialized when state $x$ reaches its limits in order to pull $x$ out of saturation immediately. You can achieve this by selecting Reinitialize dx/dt when x reaches saturation on the Attributes pane.

If this option is on, then at the instant when $x$ reaches saturation, Simulink checks whether the current value of the $dx/dt$ initial condition (parameter or signal) allows the state $x$ to leave saturation immediately. If so, Simulink reinitializes state $dx/dt$ with the value of the initial condition (parameter or signal) at that instant. If not, Simulink ignores this parameter at the current instant and sets $dx/dt$ to zero to make the block states consistent.

This parameter only applies at the time when $x$ actually reaches saturation limit. It does not apply at any future time when $x$ is being held at saturation.

Refer to the sections on limiting the states for more information. For an example, see “Simulation of a Bouncing Ball” on page 15-7.

Disregarding State Limits and External Reset for Linearization

For cases where you simplify your model by linearizing it, you can have Simulink disregard the limits of the states and the external reset by selecting Ignore state limits and the reset for linearization.

Specifying the Absolute Tolerance for the Block Outputs

By default Simulink software uses the absolute tolerance value specified in the Configuration Parameters dialog box (see “Error Tolerances for Variable-Step Solvers”) to compute the output of the integrator blocks. If this value does not provide sufficient error control, specify a more appropriate value for state $x$ in the Absolute tolerance x field.
and for state $dx/dt$ in the **Absolute tolerance dx/dt** field of the parameter dialog box. Simulink uses the values that you specify to compute the state values of the block.

### Specifying the Display of the Output Ports

You can control whether to display the $x$ or the $dx/dt$ output port using the **ShowOutput** parameter. You can display one output port or both; however, you must select at least one.

### Specifying the State Names

You can specify the name of $x$ states and $dx/dt$ states using the **StateNameX** and **StateNameDXDT** parameters. However, you must specify names for both or neither; you cannot specify names for just $x$ or just $dx/dt$. Both state names must have identical type and length. Furthermore, the number of names must evenly divide the number of states.

### Selecting All Options

When you select all options, the block icon looks like this.

![Block Icon](image)

### Ports

**Input**

- **$u$ — Input signal $u$**
  - scalar  |  vector  |  matrix
Input signal \( u \) to the integrator system, specified as a scalar, vector, or matrix.

Data Types: double

\( x_0 \) — Initial condition \( x_0 \)

scalar | vector | matrix

External signal specifying the initial condition \( x_0 \) to the integrator system. You can specify the initial condition as a scalar, vector, or matrix.

Dependencies

To enable this input port, set the Initial condition source \( x \) parameter to external.

Data Types: double

\( dx_0 \) — Initial condition \( dx_0 \)

scalar | vector | matrix

External signal specifying the initial condition \( dx_0 \) to the integrator system. You can specify the initial condition \( dx_0 \) as a scalar, vector, or matrix.

Dependencies

To enable this input port, set the Initial condition source \( dx/dt \) parameter to external.

Data Types: double

Output

\( x \) — Output signal \( x \)

scalar | vector | matrix

\( x \) state output signal, provided as a scalar, vector, or matrix.

Data Types: double

\( dx \) — Output signal \( dx \)

scalar | vector | matrix

dx state output signal, specified as a scalar, vector, or matrix.

Data Types: double
**Parameters**

*x*

**Initial condition source x** — Source of initial condition for state x

- **internal** (default) | **external**

Specify the source of the initial conditions for state x.

- **internal** — Get the initial conditions of state x from the **Initial condition x** parameter.
- **external** — Get the initial conditions of state x from an external block connected to the X₀ input port.

**Limitations**

Simulink software does not allow the initial condition of this block to be inf or NaN.

**Dependencies**

Selecting **internal** enables the **Initial condition x** parameter and removes the X₀ input port.

Selecting **external** disables the **Initial condition x** parameter and enables the X₀ input port.

**Programmatic Use**

**Block Parameter:** ICSourceX  
**Type:** character vector, string  
**Values:** 'internal' | 'external'  
**Default:** 'internal'

**Initial condition x** — Initial condition of state x

0.0 (default) | scalar | vector | matrix

Specify the initial condition of state x.

**Limitations**

Simulink software does not allow the initial condition of this block to be inf or NaN.
Dependencies

To enable this parameter, set Initial condition source x to internal.

Setting Initial condition source x to external disables this parameter and enables the \( X_0 \) input port.

Programmatic Use

Block Parameter: ICX
Type: character vector, string
Values: scalar | vector | matrix
Default: '0.0'

Limit x — Limit values of state x

off | on

Limit state \( x \) of the block to a value between the Lower limit \( x \) and Upper limit \( x \) parameters. The default value of the Second-Order Integrator block is off. The default value of the Second-Order Integrator Limited is on.

- To limit state \( x \) to a value between the Lower limit \( x \) and Upper limit \( x \) parameters, select this check box.
- To remove range limitations on state \( x \), clear this check box.

Dependencies

Selecting this check box enables the Upper limit \( x \) and Lower limit \( x \) parameters.

Programmatic Use

Block Parameter: LimitX
Type: character vector, string
Values: 'off' | 'on'
Default: 'off' (Second-Order Integrator) | 'on' (Second-Order Integrator Limited)

Upper limit \( x \) — Upper limit of state \( x \)

1 | inf | scalar | vector | matrix

Specify the upper limit of state \( x \). The default value for the Second-Order Integrator block is inf. The default value for the Second-Order Integrator Limited block is 1.

Tips

The upper saturation limit for state \( x \) must be strictly greater than the lower saturation limit.
Dependencies

To enable this parameter, select the Limit x check box.

Programmatic Use
Block Parameter: UpperLimitX
Type: character vector, string
Values: '1' | 'inf' | scalar | vector | matrix
Default: '1' (Second-Order Integrator Limited) | 'inf' (Second-Order Integrator)

Lower limit x — Lower limit of state x
θ (default) | -inf | scalar | vector | matrix

Specify the lower limit of state x. The default value for the Second-Order Integrator block is -inf. The default value for the Second-Order Integrator Limited block is θ.

Tip

The lower saturation limit for state x must be strictly less than the upper saturation limit.

Dependencies

To enable this parameter, select the Limit x check box.

Programmatic Use
Block Parameter: LowerLimitX
Type: character vector, string
Values: '0' | '-inf' | scalar | vector | matrix
Default: '0' (Second-Order Integrator Limited) | '-inf' (Second-Order Integrator)

Wrap x — Enable wrapping of x
off (default) | on

Enable wrapping of x between the Wrapped upper value x and Wrapped lower value x parameters. Enabling wrapping of x eliminates the need for zero-crossing detection, reduces solver resets, improves solver performance and accuracy, and increases simulation time span when modeling rotary and cyclic state trajectories.

If you specify Wrapped upper value x as inf and Wrapped lower value x as -inf, wrapping will never occur.

Dependencies

Selecting this check box enables Wrapped upper value x and Wrapped lower value x.
**Programmatic Use**

**Block Parameter:** `WrapX`  
**Type:** character vector, string  
**Values:** 'off' | 'on'  
**Default:** 'off'

**Wrapped upper value x — Upper value for wrapping x**  
pi (default) | scalar | vector | matrix

Specify the upper value for wrapping x.

**Dependencies**

To enable this parameter, select the **Wrap x** check box.

**Programmatic Use**

**Block Parameter:** `WrappedUpperValueX`  
**Type:** character vector, string  
**Values:** scalar | vector | matrix  
**Default:** 'pi'

**Wrapped lower value x — Lower value for wrapping x**  
-pi (default) | scalar | vector | matrix

Specify the lower value for wrapping x.

**Dependencies**

To enable this parameter, select the **Wrap x** check box.

**Programmatic Use**

**Block Parameter:** `WrappedLowerValueX`  
**Type:** character vector, string  
**Values:** scalar | vector | matrix  
**Default:** '-pi'

**Absolute tolerance x — Absolute tolerance for computing state x**  
auto (default) | -1 | scalar | vector

Specify the absolute tolerance for computing state x.

- You can enter auto, -1, a positive real scalar or vector.
- If you enter auto or -1, Simulink uses the absolute tolerance value in the Configuration Parameters dialog box (see “Solver Pane”) to compute state x.
• If you enter a real scalar value, that value overrides the absolute tolerance in the Configuration Parameters dialog box and is used for computing all $x$ states.

• If you enter a real vector, the dimension of that vector must match the dimension of state $x$. These values override the absolute tolerance in the Configuration Parameters dialog box.

**Programmatic Use**

**Block Parameter:** AbsoluteToleranceX

**Type:** character vector; string

**Values:** 'auto' | '-1' | any positive real scalar or vector

**Default:** 'auto'

**State name $x$ — Name for state $x**

'' (default) | character vector | string

Assign a unique name to state $x$.

**Tips**

• To assign a name to a single state, enter the name between quotes, for example, 'position'.

• To assign names to multiple $x$ states, enter a comma-delimited list surrounded by braces, for example, {'a', 'b', 'c'}. Each name must be unique.

• The state names apply only to the selected block.

• If you specify a state name for $x$, you must also specify a state name for $dx/dt$.

• State names for $x$ and $dx/dt$ must have identical types and lengths.

• The number of states must be evenly divided by the number of state names. You can specify fewer names than $x$ states, but you cannot specify more names than $x$ states. For example, you can specify two names in a system with four states. The first name applies to the first two states and the second name to the last two states. However, you must be consistent and apply the same scheme to the state names for $dx/dt$.

• To assign state names with a variable in the MATLAB workspace, enter the variable without quotes. A variable can be a character vector, string or a cell array.

**Programmatic Use**

**Block Parameter:** StateNameX

**Type:** character vector; string

**Values:** '' | user-defined

**Default:** ''
**dx/dt**

**Initial condition source dx/dt** — Source of initial condition for state $dx/dt$

Specify the source of initial conditions for state $dx/dt$ as internal or external.

**Limitations**

Simulink software does not allow the initial condition of this block to be $\text{inf}$ or NaN.

**Dependencies**

- Selecting internal enables the **Initial condition dx/dt** parameter and removes the $dx_0$ input port.
- Selecting external disables the **Initial condition dx/dt** parameter and enables the $dx_0$ input port.

**Programmatic Use**

**Block Parameter:** ICSourceDXDT  
**Type:** character vector  
**Values:** 'internal' | 'external'  
**Default:** 'internal'

**Initial condition dx/dt** — Initial condition of state $dx/dt$

Specify the initial condition of state $dx/dt$.

**Limitations**

Simulink software does not allow the initial condition of this block to be $\text{inf}$ or NaN.

**Dependencies**

- Setting **Initial condition source dx/dt** to internal enables this parameter.
- Setting **Initial condition source dx/dt** to external disables this parameter.

**Programmatic Use**

**Block Parameter:** ICDXDT  
**Type:** character vector  
**Values:** scalar | vector | matrix  
**Default:** '0.0'
Limit $dx/dt$ — Limit values of state $dx/dt$
off | on

Limit the $dx/dt$ state of the block to a value between the Lower limit $dx/dt$ and Upper limit $dx/dt$ parameters. The default value of the Second-Order Integrator block is off. The default value of the Second-Order Integrator Limited is on.

Tip

If you set saturation limits for $x$, then the interval defined by the Upper limit $dx/dt$ and Lower limit $dx/dt$ must contain zero.

Dependencies

Selecting this check box enables the Upper limit $dx/dt$ and Lower limit $dx/dt$ parameters.

Programmatic Use
Parameter: LimitDXDT
Type: character vector
Values: 'off' | 'on'
Default: 'off' (Second-Order Integrator) | 'on' (Second-Order Integrator Limited)

Upper limit $dx/dt$ — Upper limit of state $dx/dt$
inf (default) | scalar | vector | matrix

Specify the upper limit for state $dx/dt$.

Dependencies

If you limit $x$, then this parameter must have a strictly positive value.

To enable this parameter, select the Limit $dx/dt$ check box.

Programmatic Use
Block Parameter: UpperLimitDXDT
Type: character vector
Values: scalar | vector | matrix
Default: 'inf'

Lower limit $dx/dt$ — Lower limit of state $dx/dt$
-$\infty$ (default) | scalar | vector | matrix

Specify the lower limit for state $dx/dt$. 
**Dependencies**

If you limit \( x \), then this parameter must have a strictly negative value.

To enable this parameter, select the **Limit \( dx/dt \)** check box.

**Programmatic Use**

**Block Parameter:** `LowerLimitDXDT`  
**Type:** character vector  
**Values:** scalar | vector | matrix  
**Default:** `-inf`

**Absolute tolerance \( dx/dt \) — Absolute tolerance for computing state \( dx/dt \)**  
auto (default) | -1 | scalar | vector

Specify the absolute tolerance for computing state \( dx/dt \).

- You can enter `auto`, –1, a positive real scalar or vector.
- If you enter `auto` or –1, then Simulink uses the absolute tolerance value in the Configuration Parameters dialog box (see “Solver Pane”) to compute the \( dx/dt \) output of the block.
- If you enter a numeric value, that value overrides the absolute tolerance in the Configuration Parameters dialog box.

**Programmatic Use**

**Block Parameter:** `AbsoluteToleranceDXDT`  
**Type:** character vector, string, scalar, or vector  
**Values:** 'auto' | '-1' | any positive real scalar or vector  
**Default:** 'auto'

**State name \( dx/dt \) — Name for state \( dx/dt \)**  
' ' (default) | character vector | string

Assign a unique name to state \( dx/dt \).

**Tips**

- To assign a name to a single state, enter the name between quotes, for example, 'velocity'.
- To assign names to multiple \( dx/dt \) states, enter a comma-delimited list surrounded by braces, for example, {'a', 'b', 'c'}. Each name must be unique.
• The state names apply only to the selected block.
• If you specify a state name for $dx/dt$, you must also specify a state name for $x$.
• State names for $x$ and $dx/dt$ must have identical types and lengths.
• The number of states must be evenly divided by the number of state names. You can specify fewer names than $dx/dt$ states, but you cannot specify more names than $dx/dt$ states. For example, you can specify two names in a system with four states. The first name applies to the first two states and the second name to the last two states. However, you must be consistent and apply the same scheme to the state names for $x$.
• To assign state names with a variable in the MATLAB workspace, enter the variable without quotes. A variable can be a character vector, string, or a cell array.

**Programmatic Use**
**Block Parameter:** StateNameDXDT
**Type:** character vector, string
**Values:** ' ' | user-defined
**Default:** ' '

**Attributes**

**External reset — Reset states to their initial conditions**

none (default) | rising | falling | either

Reset the states to their initial conditions when a trigger event occurs in the reset signal.

• none — Do not reset the state to initial conditions.
• rising — Reset the state when the reset signal rises from a zero to a positive value or from a negative to a positive value.
• falling — Reset the state when the reset signal falls from a positive value to zero or from a positive to a negative value.
• either — Reset the state when the reset signal changes from zero to a nonzero value or changes sign.

**Programmatic Use**
**Block Parameter:** ExternalReset
**Type:** character vector, string
**Values:** 'none' | 'rising' | 'falling' | 'either'
**Default:** 'none'
Enable zero-crossing detection — Enable zero-crossing detection
on (default) | off

Select to enable zero-crossing detection. For more information, see “Zero-Crossing Detection”.

**Programmatic Use**
**Parameter:** ZeroCross  
**Type:** character vector, string  
**Values:** 'on' | 'off'  
**Default:** 'on'

Reinitialize dx/dt when x reaches saturation — Reset dx/dt when x reaches saturation
off (default) | on

At the instant when state x reaches saturation, reset dx/dt to its current initial conditions.

**Tip**

The dx/dt initial condition must have a value that enables x to leave saturation immediately. Otherwise, Simulink ignores the initial conditions for dx/dt to preserve mathematical consistency of block states.

**Programmatic Use**
**Block Parameter:** ReinitDXDTwhenXreachesSaturation  
**Type:** character vector, string  
**Values:** 'off' | 'on'  
**Default:** 'off'

Ignore state limits and the reset for linearization — Ignore state limits and external reset for linearization
off (default) | on

For linearization purposes, have Simulink ignore the specified state limits and the external reset.

**Programmatic Use**
**Block Parameter:** IgnoreStateLimitsAndResetForLinearization  
**Type:** character vector, string  
**Values:** 'off' | 'on'  
**Default:** 'off'
**Show output — Output ports to display**
both (default) | x | dx/dt

Specify the output ports on the block.

- **both** — Show both x and dx/dt output ports.
- **x** — Show only the x output port.
- **dx/dt** — Show only the dx/dt output port.

**Programmatic Use**

**Block Parameter:** ShowOutput  
**Type:** character vector, string  
**Values:** 'both' | 'x' | 'dxdt'  
**Default:** 'both'

**Block Characteristics**

<table>
<thead>
<tr>
<th>Data Types</th>
<th>double</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multidimensional Signals</td>
<td>No</td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>No</td>
</tr>
</tbody>
</table>

**Extended Capabilities**

**C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

Not recommended for production-quality code. Relates to resource limits and restrictions on speed and memory often found in embedded systems. The code generated can contain dynamic allocation and freeing of memory, recursion, additional memory overhead, and widely-varying execution times. While the code is functionally valid and generally acceptable in resource-rich environments, smaller embedded targets often cannot support such code.
In general, consider using the Simulink Model Discretizer to map continuous blocks into discrete equivalents that support production code generation. To start the Model Discretizer, select Analysis > Control Design > Model Discretizer. One exception is the Second-Order Integrator block because, for this block, the Model Discretizer produces an approximate discretization.

**See Also**
Integrator | Integrator Limited | Second-Order Integrator

**Topics**
“Zero-Crossing Detection”
“Error Tolerances for Variable-Step Solvers”
“Algebraic Loop Concepts”

**Introduced in R2010a**
Selector

Select input elements from vector, matrix, or multidimensional signal

Library:  
Simulink / Signal Routing  
HDL Coder / Signal Routing

Description

The Selector block generates as output selected or reordered elements of an input vector, matrix, or multidimensional signal.

Based on the value you enter for the Number of input dimensions parameter, a table of indexing settings is displayed. Each row of the table corresponds to one of the input dimensions in Number of input dimensions. For each dimension, you define the elements of the signal to work with. Specify a vector signal as a 1-D signal and a matrix signal as a 2-D signal. When you configure the Selector block for multidimensional signal operations, the block icon changes.

For example, assume a 6-D signal with a one-based index mode. The table of the Selector block dialog changes to include one row for each dimension. If you define dimensions as shown in the next table, the output is $Y = U(1:end, 2:6, [1 \ 3 \ 5], \text{Idx4:Idx4+7, Idx5, Idx6(1):Idx6(2)})$, where Idx4, Idx5, and Idx6 are the index ports for dimensions 4, 5, and 6.

<table>
<thead>
<tr>
<th>Row</th>
<th>Index Option</th>
<th>Index</th>
<th>Output Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Select all</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Starting index (dialog)</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>Index vector (dialog)</td>
<td>[1 3 5]</td>
<td></td>
</tr>
</tbody>
</table>
You can use an array of buses as an input signal to a Selector block. For details about defining and using an array of buses, see “Combine Buses into an Array of Buses”.

## Ports

### Input

**Port_1 — Input signal**  
scalar | vector | matrix | multidimensional

Input signal and source of elements to output signal.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 |  
Boolean | fixed point | enumerated | bus

**IndxN — Nth index signal**  
scalar | vector | matrix

External port specifying an index for the selection of the corresponding output element.

### Dependencies

To enable an external index port, in the corresponding row of the **Index Option** table, set **Index Option** to Index vector (port), Starting index (port), or Starting and ending indices (port).

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | enumerated
Output

Port_1 — Output signal
scalar | vector | matrix | multidimensional

Output signal generated from selected or reordered elements of input signal.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus

Parameters

Number of input dimensions — Number of dimensions of the input signal
1 (default) | integer

Specifies the number of dimensions of the input signal.

Programmatic Use
Block Parameter: NumberOfDimensions
Type: character vector
Values: integer
Default: '1'

Index mode — Index mode
One-based (default) | Zero-based

Specifies the indexing mode. If One-based is selected, an index of 1 specifies the first element of the input vector. If Zero-based is selected, an index of 0 specifies the first element of the input vector.

Programmatic Use
Block Parameter: IndexMode
Type: character vector
Values: 'One-based' | 'Zero-based'
Default: 'One-based'

Index Option — Index method for elements
Index vector (dialog) (default) | Select all | Index vector (port) |
Starting index (dialog) | Starting index (port) | Starting and ending indices (port)
Defines, by dimension, how the elements of the signal are to be indexed. From the list, select:

- **Select all**
  
  No further configuration is required. All elements are selected.

- **Index vector (dialog)**
  
  Enables the **Index** column. Enter the vector of indices of the elements. Do not select this option when the input signal is variable-sized.

- **Index vector (port)**
  
  No further configuration is required.

- **Starting index (dialog)**
  
  Enables the **Index** and **Output Size** columns. Enter the starting index of the range of elements to select in the **Index** column and the number of elements to select in the **Output Size** column. Do not select this option when the input signal is variable-sized.

- **Starting index (port)**
  
  Enables the **Output Size** column. Enter the number of elements to be selected in the **Output Size** column.

- **Starting and ending indices (port)**
  
  No further configuration is required.

Using this option results in a variable-size output signal. When you update, the output dimension is set to be the same as the input signal dimension. During execution, the output dimension is updated based on the signal feeding the index.

When logging output signal data, signals not selected are padded with NaN values.

The **Index** and **Output Size** columns appear as needed.

**Programmatic Use**

**Block Parameter:** IndexOptionArray

**Type:** character vector

**Values:** 'Select all' | 'Index vector (dialog)' | 'Index option (port)' | 'Starting index (dialog)' | 'Starting index (port)' | Starting and ending indices (port)
**Default:** 'Index vector (dialog)'

**Index — Index of elements**

1 (default) | integer

If the **Index Option** is Index vector (dialog), enter the index of each element you are interested in.

If the **Index Option** is Starting index (dialog), enter the starting index of the range of elements to be selected.

**Programmatic Use**

**Block Parameter:** `IndexParamArray`

**Type:** character vector

**Values:** cell array

**Default:** '{ }'

**Output Size — Width of the block output signal**

1 (default) | integer

Specifies the width of the block output signal.

**Programmatic Use**

**Block Parameter:** `OutputSizeArray`

**Type:** character vector

**Values:** cell array

**Default:** '{ }'

**Input port size — Width of the input signal**

3 (default) | integer

Specify the width of the block input signal for 1-D signals. Enter -1 to inherit from the driving block.

**Programmatic Use**

**Block Parameter:** `InputPortWidth`

**Type:** character vector

**Values:** integer

**Default:** '1'

**Sample time — Specify sample time as a value other than -1**

-1 (default) | scalar | vector
Specify the sample time as a value other than -1. For more information, see “Specify Sample Time”.

Dependencies

This parameter is not visible unless it is explicitly set to a value other than -1. To learn more, see “Blocks for Which Sample Time Is Not Recommended”.

Programmatic Use

**Block Parameter:** SampleTime  
**Type:** character vector  
**Values:** scalar or vector  
**Default:** '-1'

### Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>double</th>
<th>enumerated</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Direct Feedthrough</th>
<th>no</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Multidimensional Signals</th>
<th>yes</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Variable-Size Signals</th>
<th>yes</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Zero-Crossing Detection</th>
<th>no</th>
</tr>
</thead>
</table>

### Extended Capabilities

**C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.
**HDL Code Generation**
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

**HDL Architecture**
This block has a single, default HDL architecture.

**HDL Block Properties**

<table>
<thead>
<tr>
<th>General</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ConstrainedOutputPipeline</strong></td>
</tr>
<tr>
<td>Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. For more details, see “ConstrainedOutputPipeline” (HDL Coder).</td>
</tr>
</tbody>
</table>

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>InputPipeline</strong></td>
</tr>
<tr>
<td>Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “InputPipeline” (HDL Coder).</td>
</tr>
</tbody>
</table>

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>OutputPipeline</strong></td>
</tr>
<tr>
<td>Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “OutputPipeline” (HDL Coder).</td>
</tr>
</tbody>
</table>

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Native Floating Point</strong></td>
</tr>
<tr>
<td><strong>LatencyStrategy</strong></td>
</tr>
<tr>
<td>Specify whether to map the blocks in your design to inherit, Max, Min, or Zero for the floating-point operator. The default is inherit. See also “LatencyStrategy” (HDL Coder).</td>
</tr>
</tbody>
</table>

**Complex Data Support**
This block supports code generation for complex signals.

**PLC Code Generation**
Generate Structured Text code using Simulink® PLC Coder™.
Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

See Also
Assignment | Bus Selector | Switch

Introduced before R2006a
Sequence Viewer

Display messages, events, states, transitions, and functions between blocks during simulation

Library: Simulink / Messages & Events
        Simulink Test
        SimEvents
        Stateflow

Description

The Sequence Viewer block displays messages, events, states, transitions, and functions between certain blocks during simulation. The blocks that you can display are called lifeline blocks and include:

- Subsystems
- Referenced models
- Blocks that contain messages, such as Stateflow charts
- Blocks that call functions or generate events, such as Function Caller, Function-Call Generator, and MATLAB Function blocks
- Blocks that contain functions, such as Function-Call Subsystem and Simulink Function blocks

To see states, transitions, and events for lifeline blocks in a referenced model, you must have a Sequence Viewer block in the referenced model. Without a Sequence Viewer block in the referenced model, you can see only messages and functions for lifeline blocks in the referenced model.

Parameters

Time Precision for Variable Step — Adjust time increment precision
3 (default)

When using a variable step solver, change this parameter to adjust the time precision for the sequence viewer.
Programmatic Use
Block Parameter: VariableStepTimePrecision
Type: character vector
Values: '3' | scalar
Default: '3'

History — Maximum number of events to keep in viewer
5000 (default)

Programmatic Use
Block Parameter: History
Type: character vector
Values: '1000' | scalar
Default: '1000'

Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>bus</th>
<th>double</th>
<th>enumerated</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.
This block can be used for visualizing message transitions during simulation, but is not included in the generated code.

**HDL Code Generation**
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

This block displays messages during simulation when used in subsystems that generate HDL code, but it is not included in the hardware implementation.

**See Also**
“Use the Sequence Viewer Block to Visualize Messages, Events, and Entities” (SimEvents)

**Introduced in R2015b**
S-Function

Include S-function in model
Library: Simulink / User-Defined Functions

Description

The S-Function block provides access to S-functions from a block diagram. The S-function named as the S-function name parameter can be a Level-2 MATLAB or a Level-1 or Level-2 C MEX S-function (see “MATLAB S-Function Basics” for information on how to create S-functions).

Note Use the Level-2 MATLAB S-Function block to include a Level-2 MATLAB S-function in a block diagram.

The S-Function block displays the name of the specified S-function and the number of input and output ports specified by the S-function. Signals connected to the inputs must have the dimensions specified by the S-function for the inputs.

Parameters

S-function name — Name of the S-function
character array (default)

Use this parameter to specify the name of your S-function.

Programmatic Use
Block Parameter: FunctionName
Type: character vector
Value: name of the S-function
Default: 'system'
S-function parameters — Additional S-function parameters

cell array (default)

Specify the additional S-function parameters.

The function parameters can be specified as MATLAB expressions or as variables separated by commas. For example,

A, B, C, D, [eye(2,2);zeros(2,2)]

Note that although individual parameters can be enclosed in brackets, the list of parameters must not be enclosed in brackets.

Programmatic Use
Block Parameter: Parameters
Type: character vector
Value: S-function parameters
Default: ''

S-function modules — List additional files for code generation

cell array (default)

This parameter applies only if this block represents a C MEX S-function and you intend to use the Simulink Coder software to generate code from the model containing the block. If you use it, when you are ready to generate code, you must force the coder to rebuild the top model as explained in “Control Regeneration of Top Model Code” (Simulink Coder).

For more information on using this parameter, see “Specify Additional Source Files for an S-Function” (Simulink Coder).

Programmatic Use
Block Parameter: SFunctionModules
Type: character vector
Value: character vector of file names
Default: ''

Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Booleana</th>
<th>busa</th>
<th>doublea</th>
<th>fixed pointba</th>
<th>integera</th>
<th>singlea</th>
<th>stringa</th>
</tr>
</thead>
</table>

1-1808
<table>
<thead>
<tr>
<th>Direct Feedthrough</th>
<th>limited(^a)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multidimensional Signals</td>
<td>limited(^a)</td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>limited(^a)</td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>limited(^a)</td>
</tr>
</tbody>
</table>

\(^a\) Actual data type or capability support depends on block implementation.
\(^b\) See Writing Fixed-Point S-Functions for details on using fixed-point data types in S-functions.

### Extended Capabilities

**C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

- Actual code generation support depends on block implementation.
- S-functions that call into MATLAB are not supported for code generation.

**Fixed-Point Conversion**

Design and simulate fixed-point systems using Fixed-Point Designer™.

- Actual data type support depends on block implementation.
- See “Integrate External Code” (Fixed-Point Designer) for details on using fixed-point data types in S-functions.

### See Also

Level-2 MATLAB S-Function | S-Function Builder

### Topics

“Implementing S-Functions”
“Create a Basic C MEX S-Function”
“Write Level-2 MATLAB S-Functions”

Introduced before R2006a
**S-Function Builder**

Integrate C or C++ code to create S-functions

**Library:** Simulink / User-Defined Functions

---

**Description**

The S-function builder integrates new or existing C or C++ code and creates a C MEX S-function from specifications you provide. See “Build S-Functions Automatically” for detailed instructions on using the S-Function Builder block to generate an S-function.

Instances of the S-Function Builder block also serve as wrappers for generated S-functions in Simulink models. When simulating a model containing instances of an S-Function Builder block, Simulink software invokes the generated S-function in order to call your C or C++ code in the instance's `mdlStart`, `mdlOutputs`, `mdlDerivatives`, `mdlUpdate` and `mdlTerminate` methods. To learn how Simulink engine interacts with S-functions, see “Simulink Engine Interaction with C S-Functions”.

**Note** The S-Function Builder block does not support masking. However, you can mask a Subsystem block that contains an S-Function Builder block. For more information, see “Dynamic Masked Subsystem”.

---

**Ports**

**Input**

**In — Input to an S-function builder**

scalar | vector | matrix
The S-Function Builder can accept and complex, 1-D, or 2-D signals and nonvirtual buses. For each of these cases, the signals must have a data type that Simulink supports.

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus

**Output**

**Out — Output from an S-function builder**

scalar | vector | matrix

The S-Function Builder can output complex, 1-D, or 2-D signals and nonvirtual buses. For each of these cases, the signals must have a data type that Simulink supports.

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus

**Parameters**

See “S-Function Builder Dialog Box” in the online documentation for information on using the S-Function Builder block's parameter dialog box.

**Block Characteristics**

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>double</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

S-functions that call into MATLAB are not supported for code generation.

Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

See Also
Level-2 MATLAB S-Function | S-Function

Topics
“Build S-Functions Automatically”
“S-Function Builder Dialog Box”

Introduced before R2006a
Shift Arithmetic

Shift bits or binary point of signal

\[
\begin{align*}
Q_y &= Qu >> 8 \\
V_y &= Vu * 2^8 \\
E_y &= Eu
\end{align*}
\]

Library

Logic and Bit Operations

Description

Supported Shift Operations

The Shift Arithmetic block can shift the bits or the binary point of an input signal, or both.

For example, shifting the binary point on an input of data type \texttt{sfix(8)}, by two places to the right and left, gives these decimal values.

<table>
<thead>
<tr>
<th>Shift Operation</th>
<th>Binary Value</th>
<th>Decimal Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>No shift (original number)</td>
<td>11001.011</td>
<td>−6.625</td>
</tr>
<tr>
<td>Binary point shift right by two places</td>
<td>1100101.1</td>
<td>−26.5</td>
</tr>
<tr>
<td>Binary point shift left by two places</td>
<td>110.01011</td>
<td>−1.65625</td>
</tr>
</tbody>
</table>

This block performs arithmetic bit shifts on signed numbers. Therefore, the block recycles the most significant bit for each bit shift. Shifting the bits on an input of data type \texttt{sfix(8)}, by two places to the right and left, gives these decimal values.

<table>
<thead>
<tr>
<th>Shift Operation</th>
<th>Binary Value</th>
<th>Decimal Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>No shift (original number)</td>
<td>11001.011</td>
<td>−6.625</td>
</tr>
<tr>
<td>Bit shift right by two places</td>
<td>11110.010</td>
<td>−1.75</td>
</tr>
</tbody>
</table>
### Shift Operation

<table>
<thead>
<tr>
<th>Shift Operation</th>
<th>Binary Value</th>
<th>Decimal Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit shift left by two places</td>
<td>00101.100</td>
<td>5.5</td>
</tr>
</tbody>
</table>

### Data Type Support

The block supports input signals of the following data types:

<table>
<thead>
<tr>
<th>Input Signal</th>
<th>Supported Data Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>u</td>
<td>• Floating point</td>
</tr>
<tr>
<td></td>
<td>• Built-in integer</td>
</tr>
<tr>
<td></td>
<td>• Fixed point</td>
</tr>
<tr>
<td>s</td>
<td>• Floating point</td>
</tr>
<tr>
<td></td>
<td>• Built-in integer</td>
</tr>
<tr>
<td></td>
<td>• Fixed-point integer</td>
</tr>
</tbody>
</table>

The following rules determine the output data type:

<table>
<thead>
<tr>
<th>Data Type of Input</th>
<th>Output Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>u</td>
<td>Same as input u</td>
</tr>
<tr>
<td>Floating point</td>
<td></td>
</tr>
<tr>
<td>Built-in integer or fixed point</td>
<td>• Sign of u</td>
</tr>
<tr>
<td></td>
<td>• Word length of u</td>
</tr>
<tr>
<td></td>
<td>• Slope of $u \times 2^{(\text{max(binary points to shift)})}$</td>
</tr>
<tr>
<td></td>
<td>• Bias of $u \times 2^{(\text{max(binary points to shift} - \text{bits to shift})}$, for bit shifts where the direction is bidirectional or right</td>
</tr>
<tr>
<td></td>
<td>• Bias of $u \times 2^{(\text{max(binary points to shift} + \text{bits to shift})}$, for bit shifts where the direction is left</td>
</tr>
</tbody>
</table>

The block parameters support the following data types:
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Supported Data Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bits to shift: Number</td>
<td>• Built-in integer</td>
</tr>
<tr>
<td></td>
<td>• Fixed-point integer</td>
</tr>
<tr>
<td>Binary points to shift</td>
<td>• Built-in integer</td>
</tr>
<tr>
<td></td>
<td>• Fixed-point integer</td>
</tr>
</tbody>
</table>

For more information, see “Data Types Supported by Simulink” in the Simulink documentation.

**Parameters**

**Bits to shift: Source**

Specify whether to enter the bits to shift on the dialog box or to inherit the values from an input port.

**Bits to shift: Direction**

Specify the direction in which to shift bits: left, right, or bidirectional.

**Bits to shift: Number**

Specify a scalar, vector, or array of bit shift values. This parameter is available when **Bits to shift: Source** is Dialog.

<table>
<thead>
<tr>
<th>If the direction is...</th>
<th>Then...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Left or Right</td>
<td>Use positive integers to specify bit shifts.</td>
</tr>
<tr>
<td>Bidirectional</td>
<td>Use positive integers for right shifts and negative integers for left shifts.</td>
</tr>
</tbody>
</table>

**Binary points to shift**

Specify an integer number of places to shift the binary point of the input signal. A positive value indicates a right shift, while a negative value indicates a left shift.

**Diagnostic for out-of-range shift value**

Specify whether to produce a warning or error during simulation when the block contains an out-of-range shift value. Options include:

- None — Simulink software takes no action.
- **Warning** — Simulink software displays a warning and continues the simulation.
- **Error** — Simulink software terminates the simulation and displays an error

For more information, see “Simulation and Accelerator Mode Results for Out-of-Range Bit Shift Values” on page 1-1818.

**Check for out-of-range 'Bits to shift' in generated code**

Select this check box to include conditional statements in the generated code that protect against out-of-range bit shift values. This check box is available when **Bits to shift**: **Source** is **Input port**.

For more information, see “Code Generation for Out-of-Range Bit Shift Values” on page 1-1820.

**Out-of-Range Bit Shifts**

**Definition of an Out-of-Range Bit Shift**

Suppose that **WL** is the input word length. The shaded regions in the following diagram show out-of-range bit shift values for left and right shifts.

Similarly, the shaded regions in the following diagram show out-of-range bit shift values for bidirectional shifts.
The diagnostic for out-of-range bit shifts responds as follows, depending on the mode of operation:

<table>
<thead>
<tr>
<th>Mode</th>
<th>Diagnostic for out-of-range shift value</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>Do not report any warning or error.</td>
</tr>
<tr>
<td>Warning</td>
<td>Report a warning but continue simulation.</td>
</tr>
<tr>
<td>Error</td>
<td>Report an error and stop simulation.</td>
</tr>
<tr>
<td>Simulation</td>
<td></td>
</tr>
<tr>
<td>Accelerator modes and code generation</td>
<td>Has no effect.</td>
</tr>
</tbody>
</table>

**Simulation and Accelerator Mode Results for Out-of-Range Bit Shift Values**

Suppose that $U$ is the input, $WL$ is the input word length, and $Y$ is the output. The output for an out-of-range bit shift value for left shifts is as follows:
Similarly, the output for an out-of-range bit shift value for right shifts is as follows:

For bidirectional shifts, the output for an out-of-range bit shift value is as follows:
Code Generation for Out-of-Range Bit Shift Values

For the generated code, the method for handling out-of-range bit shifts depends on the setting of **Check for out-of-range ‘Bits to shift’ in generated code**.

<table>
<thead>
<tr>
<th>Check Box Setting</th>
<th>Generated Code</th>
<th>Simulation Results Compared to Generated Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selected</td>
<td>Includes conditional statements to protect against out-of-range bit shift values.</td>
<td>Simulation and Accelerator mode results match those of code generation.</td>
</tr>
</tbody>
</table>
| Cleared           | Does not protect against out-of-range bit shift values. | • For in-range values, simulation and Accelerator mode results match those of code generation.  
                      |                                                      | • For out-of-range values, the code generation results are compiler specific.        |

For right shifts on signed negative inputs, most C compilers use an arithmetic shift instead of a logical shift. Generated code for the Shift Arithmetic block depends on this compiler behavior.
Examples

Block Output for Right Bit Shifts

The following model compares the behavior of right bit shifts using the dialog box versus the block input port.

The key block parameter settings of the Constant blocks are:

<table>
<thead>
<tr>
<th>Block</th>
<th>Parameter</th>
<th>Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constant and Constant1</td>
<td>Constant value</td>
<td>124</td>
</tr>
<tr>
<td></td>
<td>Output data type</td>
<td>int8</td>
</tr>
<tr>
<td>Dynamic bit shift</td>
<td>Constant value</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>Output data type</td>
<td>Inherit: Inherit from 'Constant value'</td>
</tr>
</tbody>
</table>

The key block parameter settings of the Shift Arithmetic blocks are:

<table>
<thead>
<tr>
<th>Block</th>
<th>Parameter</th>
<th>Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit shift from dialog</td>
<td>Bits to shift: Source</td>
<td>Dialog</td>
</tr>
<tr>
<td></td>
<td>Bits to shift: Direction</td>
<td>Right</td>
</tr>
<tr>
<td></td>
<td>Bits to shift: Number</td>
<td>2</td>
</tr>
</tbody>
</table>
The top Shift Arithmetic block takes an input of 124, which corresponds to 01111100 in binary format. Shifting the number of bits two places to the right produces 00011111 in binary format. Therefore, the block outputs 31.

The bottom Shift Arithmetic block performs the same operation as the top block. However, the bottom block receives the bit shift value through an input port instead of the dialog box. By supplying this value as an input signal, you can change the number of bits to shift during simulation.

**Block Output for Binary Point Shifts**

The following model shows the effect of binary point shifts.

<table>
<thead>
<tr>
<th>Block</th>
<th>Parameter</th>
<th>Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit shift from input</td>
<td>Bits to shift: Source</td>
<td>Input port</td>
</tr>
<tr>
<td></td>
<td>Bits to shift: Direction</td>
<td>Right</td>
</tr>
</tbody>
</table>

The key block parameter settings of the Constant blocks are:

<table>
<thead>
<tr>
<th>Block</th>
<th>Parameter</th>
<th>Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constant and Constant1</td>
<td>Constant value</td>
<td>124</td>
</tr>
<tr>
<td></td>
<td>Output data type</td>
<td>int8</td>
</tr>
</tbody>
</table>

The key block parameter settings of the Shift Arithmetic blocks are:
The top Shift Arithmetic block takes an input of 124, which corresponds to 01111100 in binary format. Shifting the binary point three places to the right produces 01111100000 in binary format. Therefore, the top block outputs 995.

The bottom Shift Arithmetic block also takes an input of 124. Shifting the binary point three places to the left produces 01111.100 in binary format. Therefore, the bottom block outputs 15.5.

### Characteristics

<table>
<thead>
<tr>
<th>Characteristics</th>
<th>Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Data Types</strong></td>
<td>Double</td>
</tr>
<tr>
<td><strong>Sample Time</strong></td>
<td>Inherited from driving block</td>
</tr>
<tr>
<td><strong>Direct Feedthrough</strong></td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Multidimensional Signals</strong></td>
<td>No</td>
</tr>
<tr>
<td><strong>Variable-Size Signals</strong></td>
<td>No</td>
</tr>
<tr>
<td><strong>Zero-Crossing Detection</strong></td>
<td>No</td>
</tr>
<tr>
<td><strong>Code Generation</strong></td>
<td>Yes</td>
</tr>
</tbody>
</table>
Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

You can generate HDL code when **Bits to shift: Source** is **Dialog** or **Input port**.

**HDL Architecture**

The generated VHDL code uses the `shift_right` function and `sll` operator.

The generated Verilog code uses the `>>>` and `<<<` shift operators.

**HDL Block Properties**

<table>
<thead>
<tr>
<th>ConstraintedOutput Pipeline</th>
<th>Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. For more details, see “ConstraintedOutputPipeline” (HDL Coder).</th>
</tr>
</thead>
<tbody>
<tr>
<td>InputPipeline</td>
<td>Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “InputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td>OutputPipeline</td>
<td>Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “OutputPipeline” (HDL Coder).</td>
</tr>
</tbody>
</table>

**Complex Data Support**

This block supports code generation for complex signals.
Restrictions

When **Bits to shift: Source** is **Input port**, binary point shifting is not supported.

PLC Code Generation
Generates Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

*Introduced before R2006a*
Sign

Indicate sign of input

Library: Simulink / Math Operations
        HDL Coder / HDL Floating Point Operations
        HDL Coder / Math Operations

Description

Block Behavior for Real Inputs

For real inputs, the Sign block outputs the sign of the input:

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greater than zero</td>
<td>1</td>
</tr>
<tr>
<td>Equal to zero</td>
<td>0</td>
</tr>
<tr>
<td>Less than zero</td>
<td>-1</td>
</tr>
</tbody>
</table>

For vector and matrix inputs, the block outputs a vector or matrix where each element is the sign of the corresponding input element, as shown in this example:

Block Behavior for Complex Inputs

When the input \( u \) is a complex scalar, the block output matches the MATLAB result for:

\[
\text{sign}(u) = \frac{u}{\text{abs}(u)}
\]
When an element of a vector or matrix input is complex, the block uses the same formula that applies to scalar input, as shown in this example:

**Ports**

**Input**

**Port_1 — Input signal**
scalar | vector | matrix

Input signal whose sign will determine the output.

The block supports complex input signals only for floating-point data types, double and single.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point

**Output**

**Port_1 — Output signal**
scalar | vector | matrix

Output signal that is the sign of the input signal.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point
Parameters

Enable zero-crossing detection — Enable zero-crossing detection
on (default) | Boolean

Select to enable zero-crossing detection. For more information, see “Zero-Crossing Detection” in the Simulink documentation.

Programmatic Use
Block Parameter: ZeroCross
Type: character vector, string
Values: 'off' | 'on'
Default: 'on'

Sample time — Specify sample time as a value other than -1
-1 (default) | scalar | vector

Specify the sample time as a value other than -1. For more information, see “Specify Sample Time”.

Dependencies

This parameter is not visible unless it is explicitly set to a value other than -1. To learn more, see “Blocks for Which Sample Time Is Not Recommended”.

Programmatic Use
Block Parameter: SampleTime
Type: character vector
Values: scalar or vector
Default: '-1'

Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>double</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Variable-Size Signals | no
---|---
Zero-Crossing Detection | yes

## Extended Capabilities

### C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

### HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

#### HDL Architecture
This block has a single, default HDL architecture.

#### HDL Block Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConstrainedOutput Pipeline</td>
<td>Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. For more details, see “ConstrainedOutputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td>InputPipeline</td>
<td>Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “InputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td>OutputPipeline</td>
<td>Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “OutputPipeline” (HDL Coder).</td>
</tr>
</tbody>
</table>
Complex Data Support
This block supports code generation for complex signals.

PLC Code Generation
Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

See Also
Abs

Introduced before R2006a
Signal Builder

Create and generate interchangeable groups of signals whose waveforms are piecewise linear
Library: Simulink / Sources

Description

**Note** The Signal Builder block is not recommended. Use the Signal Editor block instead.

The Signal Builder block allows you to create interchangeable groups of piecewise linear signal sources and use them in a model. You can quickly switch the signal groups into and out of a model to facilitate testing. In the Signal Builder window, create signals and define the output waveforms. To open the window, double-click the block. See “Signal Groups”.

**Note** Use the `signalbuilder` function to create and access Signal Builder blocks programmatically.

Ports

Output

**Signal 1 — First output signal**
scalar | vector | matrix

First output signal from the signal group currently visible in the Signal Builder window.

Data Types: double | bus

**Signal n — nth output signal**
scalar | vector | matrix
nth output signal from the signal group currently visible in the Signal Builder window. n corresponds to the signal index.

Data Types: double | bus

### Block Characteristics

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Data Types</strong></td>
<td>double</td>
</tr>
<tr>
<td><strong>Direct Feedthrough</strong></td>
<td>no</td>
</tr>
<tr>
<td><strong>Multidimensional Signals</strong></td>
<td>no</td>
</tr>
<tr>
<td><strong>Variable-Size Signals</strong></td>
<td>no</td>
</tr>
<tr>
<td><strong>Zero-Crossing Detection</strong></td>
<td>yes</td>
</tr>
</tbody>
</table>

### Extended Capabilities

**C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

Not recommended for production code.

**See Also**

Signal Editor | Subsystem, Atomic Subsystem, Nonvirtual Subsystem, CodeReuse Subsystem | signalBuilderToSignalEditor | signalbuilder

**Topics**

“Signal Groups”

“Load Data with Interchangeable Scenarios”
“Scenarios”
“Signal Basics”

*Introduced before R2006a*
Signal Conversion

Convert signal to new type without altering signal values

Library: Simulink / Signal Attributes
HDL Coder / Signal Attributes

Description

The Signal Conversion block converts a signal from one type to another. To select the type of conversion to perform, use the Output parameter or one of these preconfigured Signal Conversion blocks:

- Signal Copy block
- To Virtual Bus block
- To Nonvirtual Bus block

Ports

Input

Port_1 — Input signal to convert
scalar | vector | matrix | N-D array

Input signal to convert, specified as a scalar, vector, matrix, or N-D array.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus

Output

Port_1 — Converted signal
scalar | vector | matrix | N-D array

Output signal is the input signal converted to the specified type.
Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus

Parameters

Output — Type of conversion
Signal copy (default) | Virtual bus | Nonvirtual bus

Specify the type of conversion to perform. The type of conversion that you use depends on your modeling goal.

<table>
<thead>
<tr>
<th>Modeling Goal</th>
<th>Output Option</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reduce generated code for a muxed signal.</td>
<td>Signal copy</td>
</tr>
<tr>
<td>For an example involving Simulink Coder software, see “Generate Reentrant Code from Subsystems” (Simulink Coder).</td>
<td></td>
</tr>
<tr>
<td>Connect a block with a constant sample time to an output port of an enabled subsystem.</td>
<td>Signal copy</td>
</tr>
<tr>
<td>Pass a bus, or array of buses, whose components have different data types to a nonvirtual Inport block in an atomic subsystem that has direct feedthrough.</td>
<td>Signal copy</td>
</tr>
<tr>
<td>For more information, see “Use Buses with Atomic Subsystem Nonvirtual Inport Blocks”.</td>
<td></td>
</tr>
<tr>
<td>Save memory by converting a nonvirtual bus to a virtual bus.</td>
<td>Virtual bus</td>
</tr>
<tr>
<td>Pass a virtual bus to a modeling construct that requires a nonvirtual bus, such as a Model block.</td>
<td>Nonvirtual bus</td>
</tr>
</tbody>
</table>

- The Signal copy option is the default. The type of conversion that the Signal Conversion block performs using the Signal copy option depends on the type of input signal.
<table>
<thead>
<tr>
<th>Type of Input Signal</th>
<th>Conversion That the Signal Copy Option Performs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Muxed (nonbus) signal</td>
<td>Converts the muxed signal, whose elements occupy discontiguous areas of memory, to a vector signal, whose elements occupy contiguous areas of memory. The conversion allocates a contiguous area of memory for the elements of the muxed signal and copies the values from the discontiguous areas (represented by the block input) to the contiguous areas (represented by the block output) at each time step.</td>
</tr>
<tr>
<td>Bus</td>
<td>Outputs a contiguous copy of the bus that is the input to the Signal Conversion block.</td>
</tr>
</tbody>
</table>

For an array of buses input signal, use the Signal copy option.

- The Virtual bus option converts a nonvirtual bus to a virtual bus.
- The Nonvirtual bus option converts a virtual bus to a nonvirtual bus.

**Programmatic Use**

**Block Parameter:** ConversionOutput  
**Type:** character vector  
**Values:** 'Signal copy' | 'Virtual bus' | 'Nonvirtual bus'  
**Default:** 'Signal copy'

**Data type — Nonvirtual bus data type**

Inherit: auto (default) | Bus: <object name> | <data type expression>

Specify the output data type of the nonvirtual bus that the Signal Conversion block produces.

This option is available only when you set the **Output** parameter to Nonvirtual bus.

The default option is Inherit: auto, which uses a rule that inherits a data type.

Specify a Simulink.Bus object in the **Data type** parameter for one or both of the following blocks:
- Signal Conversion block
- An upstream Bus Creator block

If you specify a bus object for the Signal Conversion block, but not for its upstream Bus Creator block, then use a bus object that matches the hierarchy of the bus that upstream Bus Creator block outputs.

If you specify a bus object for both the Signal Conversion block and its upstream Bus Creator block, use the same bus object for both blocks.

Click the Show data type assistant button to display the Data Type Assistant, which helps you set the data type attributes. For more information, see “Specify Data Types Using Data Type Assistant”.

**Dependencies**

To enable this parameter, set Output to Nonvirtual bus.

**Programmatic Use**

- **Block Parameter:** OutDataTypeStr
  - **Type:** character vector
  - **Values:** 'Inherit: auto' | 'Bus: <object name>' | '<data type expression>'
  - **Default:** 'Inherit: auto'

**Exclude this block from 'Block reduction' optimization — Exclude from block reduction optimization**

- **off** (default) | **on**

This option is available only when you set the Output parameter to Signal copy. If the elements of the input signal occupy contiguous areas of memory, then as an optimization, Simulink software eliminates the block from the compiled model. If you select the Exclude this block from 'Block reduction' optimization check box, the optimization occurs the next time you compile the model. For more information, see “Block reduction”.

**Programmatic Use**

- **Block Parameter:** OverrideOpt
  - **Type:** character vector
  - **Values:** 'off' | 'on'
  - **Default:** 'off'
Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>double</th>
<th>enumerated</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
<th>string</th>
</tr>
</thead>
</table>

| Direct Feedthrough              | yes     |
| Multidimensional Signals        | yes     |
| Variable-Size Signals           | yes     |
| Zero-Crossing Detection         | no      |

Extended Capabilities

**C/C++ Code Generation**
Generate C and C++ code using Simulink® Coder™.

**HDL Code Generation**
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

**HDL Architecture**
This block has a pass-through implementation.
HDL Block Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConstrainedOutput Pipeline</td>
<td>Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. For more details, see “ConstrainedOutputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td>InputPipeline</td>
<td>Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “InputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td>OutputPipeline</td>
<td>Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “OutputPipeline” (HDL Coder).</td>
</tr>
</tbody>
</table>

Complex Data Support

This block supports code generation for complex signals.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

Design and simulate fixed-point systems using Fixed-Point Designer™.

See Also

Bus Creator | Data Type Conversion

Topics

“Virtual Bus”

Introduced before R2006a
Signal Editor

Display, create, edit, and switch interchangeable scenarios

Library: Simulink / Sources

Description

The Signal Editor block displays, creates, and edits interchangeable scenarios. You can also use the block to switch scenarios in and out of models.

The Signal Editor block supports MAT-files that contain one or more scalar Simulink.SimulationData.Dataset objects.

You can port Signal Builder block configurations to the Signal Editor block using the signalBuilderToSignalEditor function. For more information, see “Replace Signal Builder Block with Signal Editor Block”.

To programmatically get the total number of scenarios and signals in the Signal Editor block, use the get_param NumberOfScenarios and NumberOfSignals properties. These properties contain the values as character vectors. To convert these values to doubles, use the str2double function.

To programmatically enable the Signal Editor block to label output signal lines, set the set_param PreserveSignalName to 'on' (default). For example:

set_param(gcb,'PreserveSignalName','on')

Limitations

The Signal Editor block does not support:

- Function-calls
- Array of buses
• Buses while using rapid accelerator mode
• timetable objects
• Ground signals

The Signal Editor block supports dynamic strings. It does not support strings with maximum length. In addition, strings in the Signal Editor block cannot output:

• Non-scalar MATLAB strings.
• String data that contains missing values.
• String data that contains non-ASCII characters.

Ports

Output Arguments

Signal1 — Signals in scenario
multidimensional

One or more signals, which can be:

• A MATLAB timeseries object
• A structure of MATLAB timeseries objects
• A two-dimensional matrix

The port label depends on the associated variable in the data set MAT-file.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | string | fixed point | enumerated | bus

Parameters

File name — Data set file
untitled.mat (default) | character vector

Data set file, specified as character vector, containing one or more scalar Simulink.SimulationData.Dataset objects. Do not use a file name from one locale
in a different locale. When using the block on multiple platforms, consider specifying just the MAT-file name and having the MAT-file be on the MATLAB path.

**Dependencies**

- If `untitled.mat` does not exist in the current folder, these parameters are disabled:
  - **Active scenario**
  - **Signals**
  - **Output a bus signal**
  - **Unit**
  - **Sample time**
  - **Interpolate data**
  - **Enable zero-crossing detection**
  - **Form output after final data value by**

To create a MAT-file, click ![Create MAT-file](image). This button starts the Signal Editor user interface, which lets you create and edit scenario MAT-files.

**Programmatic Use**

**Block Parameter:** Filename  
**Type:** character vector  
**Values:** character vector  
**Default:** 'untitled.mat'

**Active scenario — Active scenario**

**Scenario (default) | character vector**

Active scenario, specified as a character vector. The specified MAT-file must exist.

**Dependencies**

- To enable this parameter, ensure that the specified MAT-file exists.
- With fast restart enabled, you can:
  - Change the active scenario
  - Change the active signal
  - Start the Signal Editor user interface and edit data
While you can change the active signal, you cannot edit the signal properties in the block.

**Programmatic Use**

**Block Parameter:** ActiveScenario  
**Type:** character vector | numeric  
**Values:** character vector | index value  
**Default:** 'Scenario'

To create and edit scenarios, launch Signal Editor user interface —  
**Start Signal Editor** button click

To start Signal Editor user interface, click .

**Active signal — Signal to configure**  
**Signal 1 (default) | character vector**

Signal to configure, specified as a signal name. This signal is considered the active signal. The MAT-file must exist before you can configure signals.

To enable this parameter, ensure that the specified MAT-file exists.

**Tip** Do not use the set_param function to set the active signal property Name-Value argument ('ActiveSignal') in combination with another Name-Value pair argument for the Signal Editor block.

**Programmatic Use**

**Block Parameter:** ActiveSignal  
**Type:** character vector | numeric  
**Values:** character vector | index vector  
**Default:** 'Signal 1'

**Output a bus signal — Configure signal as bus**  
**off (default) | on**

Configure signal as a bus:

**On**  
Configure signal as a bus.
Off

Do not configure signal as a bus.

The specified MAT-file must exist.

**Tip** Select this check box only if the active signal is a MATLAB structure whose fields are the same as an existing, or create a new matching object. If you select this check box for an active signal that is not a MATLAB structure, simulation returns an error.

**Dependencies**

- Selecting **Output a bus signal** check box enables the **Select bus object** parameter.
- To enable this parameter, ensure that the specified MAT-file exists.

**Programmatic Use**

<table>
<thead>
<tr>
<th>Block Parameter:</th>
<th>IsBus</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Type:</strong></td>
<td>character vector</td>
</tr>
<tr>
<td><strong>Values:</strong></td>
<td>'off'</td>
</tr>
<tr>
<td><strong>Default:</strong></td>
<td>'off'</td>
</tr>
</tbody>
</table>

**Select bus object — Bus object**

| Bus:       | BusObject (default) | bus object name |

Select or enter the bus object name. To edit the bus object or create a bus object using the Data Type Assistant, click `>>`. The active signal must be a MATLAB structure whose fields are the same as this bus object. The specified MAT-file must exist.

**Dependencies**

To enable this parameter, ensure that the specified MAT-file exists.

**Programmatic Use**

<table>
<thead>
<tr>
<th>Block Parameter:</th>
<th>OutputBusObjectStr</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Type:</strong></td>
<td>character vector</td>
</tr>
<tr>
<td><strong>Values:</strong></td>
<td>bus object name</td>
</tr>
<tr>
<td><strong>Default:</strong></td>
<td>'Bus: BusObject'</td>
</tr>
</tbody>
</table>

**Mode — Bus object mode**

| Bus Object (default) | bus object data type |

1-1844
Select the bus object name. If you do not have a bus object, create one by clicking **Edit**, which starts the Bus Editor. For more information, see “Create and Specify Bus Objects”.

**Unit — Physical unit**

*inherit* (default) | supported physical unit

Physical unit of the signal, specified as an allowed unit. To specify a unit, begin typing in the text box. As you type, the parameter displays potential matching units. For more information, see “Unit Specification in Simulink Models”. For a list of supported units, see Allowed Unit Systems.

To constrain the unit system, click the link to the right of the parameter:

- If a Unit System Configuration block exists in the component, its dialog box opens. Use that dialog box to specify allowed and disallowed unit systems for the component.
- If a Unit System Configuration block does not exist in the component, the model Configuration Parameters dialog box displays. Use that dialog box to specify allowed and disallowed unit systems for the model.

The specified MAT-file must exist.

**Dependencies**

To enable this parameter, ensure that the specified MAT-file exists.

**Programmatic Use**

**Block Parameter:** Unit

**Type:** character vector

**Values:** 'inherit' | supported physical unit

**Default:** 'inherit'

**Sample time — Time interval between samples**

0 (default) | -1 | sample time in seconds

Time interval between samples, specified in seconds. The specified MAT-file must exist.

**Dependencies**

To enable this parameter, ensure that the specified MAT-file exists.

**Programmatic Use**

**Block Parameter:** SampleTime

**Type:** character vector
Values: -1 | sample time in seconds

Default: '0'

Interpolate data — Linearly interpolate data

off (default) | on

Linearly interpolate data at time hits for which no corresponding workspace data exist. The specified MAT-file must exist.

The Signal Editor block linearly interpolates:

- **On**
  - Linearly interpolate at time hits for which no corresponding workspace data exist, select this option.

- **Off**
  - The current output equals the output at the most recent time for which data exists.

The Signal Editor block interpolates by using the two corresponding workspace samples:

- For **double** data, linearly interpolates the value by using the two corresponding samples
- For **Boolean** data, uses false for the first half of the time between two time values and true for the second half
- For a built-in data type other than **double** or **Boolean**:
  - Upcasts the data to **double**
  - Performs linear interpolation (as described for **double** data)
  - Downcasts the interpolated value to the original data type

You cannot use linear interpolation with enumerated (**enum**) data.

The block uses the value of the last known data point as the value of time hits that occur after the last known data point.

To determine the block output after the last time hit for which data is available, combine the settings of these parameters:

- Interpolate data
• Form output after final data value by

For details, see the **Form output after final data value by** parameter.

**Dependencies**

To enable this parameter, ensure that the specified MAT-file exists.

**Programmatic Use**

**Block Parameter:** Interpolate

**Type:** character vector

**Values:** 'off' | 'on'

**Default:** 'off'

**Enable zero-crossing detection — Detect zero-crossings**

**off** (default) | **on**

If you select the **Enable zero-crossing detection** parameter, then when the input array contains multiple entries for the same time hit, Simulink detects a zero crossing. For example, suppose that the input array has this data:

```
time:     0 1 2 2 3
signal:   2 3 4 5 6
```

At time 2, there is a zero crossing from input signal discontinuity. For more information, see “Zero-Crossing Detection”.

For bus signals, Simulink detects zero crossings across all leaf bus elements.

The specified MAT-file must exist.

**Dependencies**

To enable this parameter, ensure that the specified MAT-file exists.

**Programmatic Use**

**Block Parameter:** ZeroCross

**Type:** character vector

**Values:** 'off' | 'on'

**Default:** 'off'

**Form output after final data value by — Block output after the last time hit for which data is available**

Setting to zero (default) | **Extrapolation** | **Holding final value**
To determine the block output after the last time hit for which workspace data is available, combine the settings of these parameters:

- **Interpolate data**
- **Form output after final data value by**

This table lists the block output, based on the values of the two options.

<table>
<thead>
<tr>
<th>Setting for Form Output After Final Data Value By</th>
<th>Setting for Interpolate Data</th>
<th>Block Output After Final Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extrapolation</td>
<td>On</td>
<td>Extrapolated from final data value</td>
</tr>
<tr>
<td></td>
<td>Off</td>
<td>Error</td>
</tr>
<tr>
<td>Setting to zero</td>
<td>On</td>
<td>Zero</td>
</tr>
<tr>
<td></td>
<td>Off</td>
<td>Zero</td>
</tr>
<tr>
<td>Holding final value</td>
<td>On</td>
<td>Final value from workspace</td>
</tr>
<tr>
<td></td>
<td>Off</td>
<td>Final value from workspace</td>
</tr>
</tbody>
</table>

For example, the block uses the last two known data points to extrapolate data points that occur after the last known point if you:

- Select **Interpolate data**.
- Set **Form output after final data value by** to **Extrapolation**.

The specified MAT-file must exist.

**Dependencies**

To enable this parameter, ensure that the specified MAT-file exists.

**Programmatic Use**

**Block Parameter:** OutputAfterFinalValue  
**Type:** character vector  
**Values:** 'Setting to zero' | 'Extrapolation' | 'Holding final value'  
**Default:** 'Setting to zero'
Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>bus</th>
<th>double</th>
<th>enumerated</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
<th>string</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Extended Capabilities

**C/C++ Code Generation**
Generate C and C++ code using Simulink® Coder™.

Not recommended for production code.

**Fixed-Point Conversion**
Design and simulate fixed-point systems using Fixed-Point Designer™.

See Also
Signal Builder | signalBuilderToSignalEditor | signalEditor | str2double

Topics
“Create and Specify Bus Objects”
“Load Data with Interchangeable Scenarios”
“Create and Edit Signal Data”
Introduced in R2017b
Signal Generator

Generate various waveforms

Library: Simulink / Sources

Description

Supported Operations

The Signal Generator block can produce one of four different waveforms:

- sine
- square
- sawtooth
- random

You can express signal parameters in hertz or radians per second. Using default parameter values, you get one of the following waveforms:
<table>
<thead>
<tr>
<th>Waveform</th>
<th>Scope Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sine wave</td>
<td><img src="image" alt="Sine Wave Graph" /></td>
</tr>
<tr>
<td>Waveform</td>
<td>Scope Output</td>
</tr>
<tr>
<td>--------------</td>
<td>-------------</td>
</tr>
<tr>
<td>Square wave</td>
<td>![Scope Output Image]</td>
</tr>
</tbody>
</table>

![Scope Output Image]
<table>
<thead>
<tr>
<th>Waveform</th>
<th>Scope Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sawtooth wave</td>
<td><img src="image" alt="Sawtooth wave graph" /></td>
</tr>
</tbody>
</table>
A negative **Amplitude** parameter value causes a 180-degree phase shift. You can generate a phase-shifted wave at other than 180 degrees in many ways. For example, you can connect a Clock block signal to a MATLAB Function block and write the equation for the specific wave.

You can vary the output settings of the Signal Generator block while a simulation is in progress to determine quickly the response of a system to different types of inputs.

The **Amplitude** and **Frequency** parameters determine the amplitude and frequency of the output signal. The parameters must be of the same dimensions after scalar expansion. If you clear the **Interpret vector parameters as 1-D** check box, the block outputs a signal of the same dimensions as the **Amplitude** and **Frequency** parameters (after scalar expansion). If you select the **Interpret vector parameters as 1-D** check box, the block outputs a vector (1-D) signal if the **Amplitude** and **Frequency** parameters are row or column vectors, that is, single-row or column 2-D arrays. Otherwise, the block outputs a signal of the same dimensions as the parameters.
Solver Considerations

If your model uses a fixed-step solver, Simulink uses the same step size for the entire simulation. In this case, the Signal Generator block output provides a uniformly sampled representation of the ideal waveform.

If your model uses a variable-step solver, Simulink might use different step sizes during the simulation. In this case, the Signal Generator block output does not always provide a uniformly sampled representation of the ideal waveform. To ensure that the block output is a uniformly sampled representation, add a Hit Crossing block directly downstream of the Signal Generator block. These models show the difference in Signal Generator block output with and without the Hit Crossing block.

<table>
<thead>
<tr>
<th>Model That Uses a Variable-Step Solver</th>
<th>Signal Generator Block Output</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Signal Generator Block" /></td>
<td><img src="image" alt="Scope Output" /></td>
</tr>
</tbody>
</table>

1-1856
Ports

Output

Port_1 — Generated output signal
scalar | vector | matrix

Output signal specified as one of these waveforms.
• sine
• square
• sawtooth
• random

Data Types: double

Parameters

Wave form — Wave form to generate
sine (default) | square | sawtooth | random

Specify the wave form.

Programmatic Use
Block Parameter: WaveForm
Type: character vector
Values: 'sine' | 'square' | 'sawtooth' | 'random'
Default: 'sine'

Time (t) — Source of time variable
Use simulation time (default) | Use external signal

Specify whether to use simulation time or an external signal as the source of values for the waveform time variable. If you specify an external source, the block displays an input port for connecting the source.

Programmatic Use
Block Parameter: TimeSource
Type: character vector
Values: 'Use simulation time' | 'Use external signal'
Default: 'Use simulation time'

Amplitude — Signal amplitude
1 (default) | scalar | vector | matrix

Specify the amplitude of the generated waveform.

Programmatic Use
Block Parameter: Amplitude
Type: character vector
Values: real scalar
Default: '1'

**Frequency — Signal frequency**
1 (default) | scalar | vector | matrix

Specify the frequency of the generated waveform.

**Programmatic Use**
**Block Parameter:** Frequency
Type: character vector
Values: real scalar
Default: '1'

**Units — Signal units**
rad/sec (default) | Hertz

Specify the signal units as Hertz or rad/sec.

**Programmatic Use**
**Block Parameter:** Units
Type: character vector
Values: 'rad/sec' | 'Hertz'
Default: 'rad/sec'

**Interpret vector parameters as 1-D — Treat vectors as 1-D**
on (default) | off

Select this check box to output a vector of length N if the **Constant value** parameter evaluates to an N-element row or column vector.

- When you select this check box, the block outputs a vector of length N if the **Constant value** parameter evaluates to an N-element row or column vector. For example, the block outputs a matrix of dimension 1-by-N or N-by-1.
- When you clear this check box, the block does not output a vector of length N if the **Constant value** parameter evaluates to an N-element row or column vector.

**Programmatic Use**
**Block Parameter:** VectorParams1D
Type: character vector
Values: 'on' | 'off'
Default: 'on'

Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>double</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>yes</td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>yes</td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
</tr>
</tbody>
</table>

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

Cannot be used inside a triggered subsystem hierarchy.

These blocks do not reference absolute time when configured for sample-based operation. In time-based operation, they depend on absolute time.

See Also
Pulse Generator | Signal Editor | Waveform Generator

Topics
“Scenarios”
Introduced before R2006a
Signal Specification

Specify desired dimensions, sample time, data type, numeric type, and other attributes of signal

Library: Simulink / Signal Attributes
        HDL Coder / Signal Attributes

Description

The Signal Specification block allows you to specify the attributes of the signal connected to its input and output ports. If the specified attributes conflict with the attributes specified by the blocks connected to its ports, Simulink software displays an error when it compiles the model. For example, at the beginning of a simulation, if no conflict exists, Simulink eliminates the Signal Specification block from the compiled model. In other words, the Signal Specification block is a virtual block. It exists only to specify the attributes of a signal and plays no role in the simulation of the model.

You can use the Signal Specification block to ensure that the actual attributes of a signal meet desired attributes. For example, suppose that you and a colleague are working on different parts of the same model. You use Signal Specification blocks to connect your part of the model with your colleague’s. If your colleague changes the attributes of a signal without informing you, the attributes entering the corresponding Signal Specification block do not match. When you try to simulate the model, you get an error.

You can also use the Signal Specification block to ensure correct propagation of signal attributes throughout a model. However, if some blocks have unspecified attributes for the signals they accept or output, the model does not have enough information to propagate attributes correctly. In these cases, the Signal Specification block can provide the information Simulink needs. Using the Signal Specification block also helps speed up model compilation when blocks are missing signal attributes.

The Signal Specification block supports signal label propagation.
**Ports**

**Input**

**Port_1 — Input signal**

scalar | vector | matrix | N-D array

Input signal whose attributes to check, specified as a scalar, vector, matrix, or N-D array. The block checks the attributes of the input signal against the desired attributes you specify in the block dialog box. If the attributes do not match, the block generates an error.

**Data Types:** single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus

**Output**

**Port_1 — Output signal**

scalar | vector | matrix | N-D array

Output signal is the input signal when all attributes of the signal match those specified in the dialog box. If the attributes do not match, the block generates an error.

**Data Types:** single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus

**Parameters**

**Minimum — Minimum value for range checking**

[] (default) | scalar

Specify the minimum value for the block output as a finite real double scalar value.

**Note** If you specify a bus object as the data type for this block, do not set the minimum value for bus data on the block. Simulink ignores this setting. Instead, set the minimum values for bus elements of the bus object specified as the data type. For information on the Minimum property of a bus element, see Simulink.BusElement.
Simulink uses this value to perform:

- Simulation range checking (see “Signal Ranges”)
- Automatic scaling of fixed-point data types

**Programmatic Use**

**Block Parameter:** `OutMin`  
**Type:** character vector  
**Values:** scalar  
**Default:** `'[]'`

**Maximum — Maximum value for range checking**

`[]` (default) | scalar

Specify the maximum value for the block output as a finite real double scalar value.

**Note** If you specify a bus object as the data type for this block, do not set the maximum value for bus data on the block. Simulink ignores this setting. Instead, set the maximum values for bus elements of the bus object specified as the data type. For information on the Maximum property of a bus element, see `Simulink.BusElement`.

Simulink uses this value to perform:

- Simulation range checking (see “Signal Ranges”)
- Automatic scaling of fixed-point data types

**Programmatic Use**

**Block Parameter:** `OutMax`  
**Type:** character vector  
**Values:** scalar  
**Default:** `'[]'`

**Data type — Output data type**

`Inherit: auto` (default) | `double` | `single` | `int8` | `uint8` | `int16` | `uint16` | `int32` | `uint32` | `int64` | `uint64` | `boolean` | `fixdt(1,16)` | `fixdt(1,16,0)` | `fixdt(1,16,2^0,0)` | `Enum: <class name>` | `Bus: <object name>` | `<data type expression>`

Specify the desired output data type. If the data type of the input signal does not match the value you specify, the block generates an error.
Click the **Show data type assistant** button to display the **Data Type Assistant**, which helps you set the data type attributes. For more information, see “Specify Data Types Using Data Type Assistant”.

**Programmatic Use**

**Block Parameter:** OutDataTypeStr  
**Type:** character vector  
**Value:** 'Inherit: auto' | 'double' | 'single' | 'int8' | 'uint8' | 'int16' | 'uint16' | 'int32' | 'uint32' | 'int64' | 'uint64' | 'boolean' | 'fixdt(1,16,0)' | 'fixdt(1,16,2^0,0)' | 'Enum: <class name>' | 'Bus: <object name>' | <data type expression>  
**Default:** 'Inherit: auto'

**Lock output data type setting against changes by the fixed-point tools — Prevent fixed-point tools from overriding Output data type**

**off** (default) | **on**

Select this parameter to prevent the fixed-point tools from overriding the **Output** data type you specify on the block. For more information, see “Use Lock Output Data Type Setting” (Fixed-Point Designer).

**Programmatic Use**

**Block Parameter:** LockScale  
**Type:** character vector  
**Values:** 'off' | 'on'  
**Default:** 'off'

**Require nonvirtual bus — Accept only nonvirtual bus signals**

**off** (default) | **on**

If you specify a bus object as the data type, use this parameter to specify whether to accept only nonvirtual bus signals.

- **off** — Specifies that a signal must come from a **virtual** bus.
- **on** — Specifies that a signal must come from a **nonvirtual** bus.

**Dependencies**

To enable this parameter, set **Data type** to one of these values:

- **Bus:** <object name>
• `<data type expression>` that specifies a bus object

**Programmatic Use**

**Block Parameter:** BusOutputAsStruct  
**Type:** character vector  
**Value:** 'off' | 'on'  
**Default:** 'off'

**Unit (e.g., m, m/s\(^2\), N*m) — Physical unit of the input signal to the block**

Specify the physical unit of the input signal to the block. To specify a unit, begin typing in the text box. As you type, the parameter displays potential matching units. For a list of supported units, see Allowed Unit Systems.

To constrain the unit system, click the link to the right of the parameter:

• If a Unit System Configuration block exists in the component, its dialog box opens. Use that dialog box to specify allowed and disallowed unit systems for the component.

• If a Unit System Configuration block does not exist in the component, the model Configuration Parameters dialog box displays. Use that dialog box to specify allowed and disallowed unit systems for the model.

**Programmatic Use**

**Block Parameter:** Unit  
**Type:** character vector  
**Values:** 'inherit' | '<Enter unit>'  
**Default:** 'inherit'

**Dimensions (-1 for inherited) — Dimensions of input and output signals**

-1 (default) | n | [m n]

Specify the dimensions of the input and output signals.

• -1 — Specifies that signals inherit dimensions.

• n — Specifies a vector of width n.

• [m n] — Specifies a matrix with m rows and n columns.

**Programmatic Use**

**Block Parameter:** Dimensions  
**Type:** character vector
Values: '-1' | n | [m n]
Default: '-1'

Variable-size signal — Allow signal to be variable-size, fixed-size, or both
Inherit (default) | No | Yes

Specify the signal to be of variable-size, fixed size, or both.
• Inherit — Allows variable-size and fixed-size signals.
• No — Does not allow variable-size signals.
• Yes — Allows only variable-size signals.

Dependencies

When the signal is a variable-size signal, the Dimensions parameter specifies the maximum dimensions of the signal.

If you specify a bus object, the simulation allows variable-size signals only with a disabled bus object.

Programmatic Use
Block Parameter: VarSizeSig
Type: character vector
Values: 'Inherit' | 'No' | 'Yes'
Default: 'Inherit'

Sample time (-1 for inherited) — Time interval between samples
-1 (default) | scalar | vector

Specify the time interval between samples. To inherit the sample time, set this parameter to -1. See “Specify Sample Time” for more information.

Programmatic Use
Parameter: SampleTime
Type: character vector
Values: Any valid sample time
Default: '-1'

Signal type — Complexity of signal
auto (default) | real | complex

Specify the complexity of the input and output signals.
- auto — Accepts either real or complex as the numeric type.
- real — Specifies the numeric type as a real number.
- complex — Specifies the numeric type as a complex number.

**Programmatic Use**

**Parameter:** SignalType  
**Type:** character vector  
**Values:** 'auto' | 'real' | 'complex'  
**Default:** 'auto'

**Block Characteristics**

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>bus</th>
<th>double</th>
<th>enumerated</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
<th>string</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Extended Capabilities**

**C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

**HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.
HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

**HDL Architecture**

This block has a pass-through implementation.

**HDL Block Properties**

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ConstrainedOutputPipeline</strong></td>
<td>Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. For more details, see “ConstrainedOutputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td><strong>InputPipeline</strong></td>
<td>Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “InputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td><strong>OutputPipeline</strong></td>
<td>Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “OutputPipeline” (HDL Coder).</td>
</tr>
</tbody>
</table>

**Complex Data Support**

This block supports code generation for complex signals.

**Fixed-Point Conversion**

Design and simulate fixed-point systems using Fixed-Point Designer™.

**See Also**

Unit System Configuration

**Topics**

“Unit Specification in Simulink Models”
“Variable-Size Signal Basics”
“Specify Sample Time”
Introduced before R2006a
**Simulink Function**

Function defined with Simulink blocks  
**Library:**  
Simulink / User-Defined Functions  
SimEvents

**Description**

The Simulink Function block is a Subsystem block preconfigured as a starting point for graphically defining a function with Simulink blocks. The block provides a text interface to function callers. You can call a Simulink Function block from a Function Caller block, a MATLAB Function block, or a Stateflow Chart.

For a description of the block parameters, see the Subsystem block reference page in the Simulink documentation.

You can visualize Simulink Function calls in the Sequence Viewer. The viewer shows when calls were made with the argument and the return values. See Sequence Viewer block reference.

**Function Interface**

The function interface appears on the face of a Simulink Function block. Editing the block text adds and deletes Argument Inport blocks and Argument Outport blocks from the function definition. Editing also sets the **Function name** parameter in the Trigger block within the Simulink Function block.

For example, entering \( y = \text{myfunction}(u) \) on the face of a Simulink Function block adds one Argument Inport block \( (u) \) and one Argument Outport block \( (y) \) within the subsystem.

When calling a function using a Function Caller block, the parameter **Function prototype** in the Function Caller block must match exactly the function interface you specify on the Simulink Function block. This match includes the name of the function and the names of input and output arguments. For example, the Simulink Function block and the Function Caller block both use the argument names \( u \) and \( y \).
When calling a function from a Stateflow transition or state label, you can use different argument names. For example, the Simulink Function block uses $x$ and $y$ arguments while the Stateflow transition uses $x_2$ and $y_2$ arguments to call the function.

**Function-Call Subsystems Versus Simulink Function Blocks**

In general, a Function-Call Subsystem block provides better signal traceability with direct signal connections than a Simulink Function block. While a Simulink Function block eliminates the need for routing input and output signal lines through the model hierarchy.
### Attribute

<table>
<thead>
<tr>
<th>Method of executing/invoking function</th>
<th>Function-Call Subsystem block</th>
<th>Simulink Function block</th>
</tr>
</thead>
<tbody>
<tr>
<td>Triggered using a signal line</td>
<td></td>
<td>Called by reference using the function name</td>
</tr>
<tr>
<td>Formal input arguments (Argument Inport blocks) and output arguments (Argument Outport blocks)</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Local inputs (Inport block) and outputs (Outport block)</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

### Ports

**Input**

**In — Signal input to a subsystem block**

*scalar | vector | matrix*

Placing an Inport block in a subsystem block adds an external input port to the Simulink Function block. The port label matches the name of the Inport block.

Use Inport blocks to get signals from the local environment.

Data Types: *single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus | struct*

**ArgIn — Argument input to a subsystem block**

*scalar | vector | matrix*

An Argument Inport block in a subsystem block provides an input port corresponding to an input argument. A port is not displayed on the subsystem block.

Data Types: *single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus | struct*

### Output

**Out — Signal output from a subsystem**

*scalar | vector | matrix*
Placing an Outport block in a subsystem block adds an output port from the block. The port label on the subsystem block is the name of the Outport block.

Use Outport blocks to send signals to the local environment.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus | struct

**ArgOut — Argument output from a subsystem block**

scalar | vector | matrix

An Argument Outport block in a subsystem block provides an output port corresponding to an output argument. A port is not displayed on the subsystem block.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus | struct

---

**Block Characteristics**

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean a</th>
<th>bus a</th>
<th>double a</th>
<th>enumerated a</th>
<th>fixed point a</th>
<th>integer a</th>
<th>single a</th>
<th>string a</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>limited a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

a. Actual data type or capability support depends on block implementation.

---

**Extended Capabilities**

**C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.
Actual code generation support depends on block implementation.

**Fixed-Point Conversion**
Design and simulate fixed-point systems using Fixed-Point Designer™.

Actual data type support depends on block implementation.

**See Also**

**Blocks**
Argument Inport | Argument Outport | Chart | Function Caller | Function-Call Subsystem | Inport | MATLAB Function | Outport | Subsystem | Trigger

**Topics**
“Simulink Functions Overview”
“Scoped Simulink Function Blocks in Models”
“Simulink functions: Simulink Function block, exported Stateflow graphical and MATLAB functions”

**Introduced in R2014b**
Sine, Cosine

Implement fixed-point sine or cosine wave using lookup table approach that exploits quarter wave symmetry  
**Library:** Simulink / Lookup Tables

![Sine block](image)

![Cosine block](image)

**Description**

The Sine and Cosine block implements a sine and/or cosine wave in fixed point using a lookup table method that exploits quarter wave symmetry. The block can output the following functions of the input signal, depending upon what you select for the **Output formula** parameter:

- \( \sin(2\pi u) \)
- \( \cos(2\pi u) \)
- \( \exp(j2\pi u) \)
- \( \sin(2\pi u) \) and \( \cos(2\pi u) \)

You define the number of lookup table points in the **Number of data points for lookup table** parameter. The block implementation is most efficient when you specify the lookup table data points to be \((2^n)+1\), where \(n\) is an integer.

Use the **Output word length** parameter to specify the word length of the fixed-point output data type. The fraction length of the output is the output word length minus 2.
Tip To simulate a model containing this block without a Fixed-Point Designer license, you must use data type override. For more information, see “Share Fixed-Point Models”.

Ports

Input

\( u \) — Input signal to implement as fixed-point sine or cosine wave

real-valued signal

Input signal, \( u \), specified as a real-valued scalar, vector, or matrix.

Tip To obtain meaningful block output, the block input values should fall within the range \([0, 1)\). For input values that fall outside this range, the values are cast to an unsigned data type, where overflows wrap. For these out-of-range inputs, the block output might not be meaningful.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point

Output

\( \sin(2\pi u) \) — Fixed-point sine wave

real-valued fixed-point signal

Fixed-point sine wave, implemented using a lookup table approach.

Dependencies

This port is enabled when the Output formula is set to \( \sin(2\pi u) \) or \( \sin(2\pi u) \) and \( \cos(2\pi u) \).

Data Types: fixed point

\( \cos(2\pi u) \) — Fixed-point cosine wave

real-valued fixed-point signal

Fixed-point cosine wave, implemented using a lookup table approach.
Dependencies

This port is enabled when the **Output formula** is set to \( \cos(2\pi u) \) or \( \sin(2\pi u) \) and \( \cos(2\pi u) \).

Data Types: fixed point

\( \exp(j2\pi u) - \exp(j2\pi u) \)

complex-valued fixed-point signal

\( \exp(j2\pi u) \), implemented using a lookup table approach.

Dependencies

This port is enabled when the **Output formula** is set to \( \exp(j2\pi u) \).

Data Types: fixed point

Parameters

**Output formula** — Select the signal(s) to output

\( \cos(2\pi u) \) | \( \sin(2\pi u) \) | \( \exp(j2\pi u) \) | \( \sin(2\pi u) \) and \( \cos(2\pi u) \)

**Programmatic Use**

Block Parameter: Formula

Values: '\( \sin(2\pi u) \)' | '\( \cos(2\pi u) \)' | '\( \exp(j2\pi u) \)' | '\( \sin(2\pi u) \) and \( \cos(2\pi u) \)'

**Number of data points for lookup table** — Specify the number of data points to retrieve from the lookup table

(\( 2^5 \))+1 (default) | integer, greater than or equal to 2

The implementation is most efficient when you specify the lookup table data points to be \( (2^n) + 1 \), where \( n \) is an integer. To be compatible with the **Output word length** parameter, the **Number of data points for lookup table** must be less than or equal to \( (2^{(\text{Output word length}-2)}+1) \).

**Programmatic Use**

Block Parameter: NumDataPoints

Type: scalar

Value: integer >= 2

Default: '\( (2^5)+1 \)'
Output word length — Specify the word length for the fixed-point data type of the output signal
16 (default) | integer from 2 to 53

The fraction length of the output is the output word length minus 2. To be compatible with the Number of data points for lookup table parameter, \((2^{(\text{Output word length} - 2)} + 1)\) must be greater than or equal to Number of data points for lookup table.

**Note** The block uses double-precision floating-point values to construct lookup tables. Therefore, the maximum amount of precision you can achieve in your output is 53 bits. Setting the word length to values greater than 53 bits does not improve the precision of your output.

**Programmatic Use**
**Block Parameter:** OutputWordLength  
**Type:** scalar  
**Value:** integer from 2 to 53  
**Default:** '16'

**Internal rule priority for lookup table — Specify the internal rule for intermediate calculations**  
Speed (default) | Precision

Select Speed for faster calculations. If you do, a loss of accuracy might occur, usually up to 2 bits.

**Programmatic Use**
**Block Parameter:** InternalRulePriority  
**Values:** 'Speed' | 'Precision'  
**Default:** 'Speed'

### Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>double</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Multidimensional Signals | no
| Variable-Size Signals | no
| Zero-Crossing Detection | no

**Extended Capabilities**

**C/C++ Code Generation**  
Generate C and C++ code using Simulink® Coder™.

**HDL Code Generation**  
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

**HDL Architecture**

The HDL code implements Cosine and Sine blocks by using the quarter-wave lookup table that you specify in the Simulink block parameters.

To avoid generating a division operator (/) in the HDL code, for **Number of data points for lookup table**, enter \((2^n) + 1\). \(n\) is an integer.

**HDL Block Properties**

| ConstrainedOutput Pipeline | Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. For more details, see “ConstrainedOutputPipeline” (HDL Coder). |
InputPipeline | Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “InputPipeline” (HDL Coder).

OutputPipeline | Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “OutputPipeline” (HDL Coder).

Limitations

If you use Intel MAX 10 device, to map the lookup table to RAM, add this Tcl command when creating the project in the Quartus tool:

```
set_global_assignment -name INTERNAL_FLASH_UPDATE_MODE "SINGLE IMAGE WITH ERAM"
```

Fixed-Point Conversion

Design and simulate fixed-point systems using Fixed-Point Designer™.

See Also

Cosine HDL Optimized | Sine HDL Optimized | Sine Wave | Trigonometric Function

Topics

“About Lookup Table Blocks”

Introduced before R2006a
Sine Wave

Generate sine wave, using simulation time as time source
Library: Simulink / Sources

Description

The Sine Wave block outputs a sinusoidal waveform. The block can operate in time-based or sample-based mode.

Note This block is the same as the Sine Wave Function block that appears in the Math Operations library. If you select Use external signal for the Time parameter in the block dialog box, you get the Sine Wave Function block.

Time-Based Mode

The block calculates the output waveform.

\[ y = \text{amplitude} \times \sin(\text{frequency} \times \text{time} + \text{phase}) + \text{bias}. \]

In time-based mode, the value of the Sample time parameter determines whether the block operates in continuous mode or discrete mode.

- \( \theta \) (the default) causes the block to operate in continuous mode.
- \( >\theta \) causes the block to operate in discrete mode.

For more information, see “Specify Sample Time”.

Block Behavior in Continuous Mode

When operating in continuous mode, the Sine Wave block can become inaccurate due to loss of precision as time becomes very large.
Block Behavior in Discrete Mode

A **Sample time** parameter value greater than zero causes the block to behave as if it were driving a Zero-Order Hold block whose sample time is set to that value.

This way, you can build models with sine wave sources that are purely discrete, rather than models that are hybrid continuous/discrete systems. Hybrid systems are inherently more complex and as a result take more time to simulate.

In discrete mode, this block uses a differential incremental algorithm instead of one based on absolute time. As a result, the block can be useful in models intended to run for an indefinite length of time, such as in vibration or fatigue testing.

The differential incremental algorithm computes the sine based on the value computed at the previous sample time. This method uses the following trigonometric identities:

\[
\sin(t + \Delta t) = \sin(t)\cos(\Delta t) + \sin(\Delta t)\cos(t)
\]

\[
\cos(t + \Delta t) = \cos(t)\cos(\Delta t) - \sin(t)\sin(\Delta t)
\]

In matrix form, these identities are:

\[
\begin{bmatrix}
\sin(t + \Delta t) \\
\cos(t + \Delta t)
\end{bmatrix} =
\begin{bmatrix}
\cos(\Delta t) & \sin(\Delta t) \\
-\sin(\Delta t) & \cos(\Delta t)
\end{bmatrix}
\begin{bmatrix}
\sin(t) \\
\cos(t)
\end{bmatrix}
\]

Because \( \Delta t \) is constant, the following expression is a constant:

\[
\begin{bmatrix}
\cos(\Delta t) & \sin(\Delta t) \\
-\sin(\Delta t) & \cos(\Delta t)
\end{bmatrix}
\]

Therefore, the problem becomes one of a matrix multiplication of the value of \( \sin(t) \) by a constant matrix to obtain \( \sin(t + \Delta t) \).

Discrete mode reduces but does not eliminate the accumulation of round-off errors, for example, \((4*\text{eps})\). This accumulation can happen because computation of the block output at each time step depends on the value of the output at the previous time step.

**Methods to Handle Round-Off Errors in Discrete Mode**

To handle round-off errors when the Sine Wave block operates in time-based discrete mode, use one of these methods.
## Method

<table>
<thead>
<tr>
<th>Method</th>
<th>Rationale</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert a Saturation block directly downstream of the Sine Wave block.</td>
<td>By setting saturation limits on the Sine Wave block output, you can remove overshoot due to accumulation of round-off errors.</td>
</tr>
<tr>
<td>Set up the Sine Wave block to use the ( \sin() ) math library function to calculate block output.</td>
<td>The ( \sin() ) math library function computes block output at each time step independently of output values from other time steps, preventing the accumulation of round-off errors.</td>
</tr>
<tr>
<td>1  On the Sine Wave block dialog box, set <strong>Time</strong> to <strong>Use external signal</strong> so that an input port appears on the block icon.</td>
<td></td>
</tr>
<tr>
<td>2  Connect a clock signal to this input port using a Digital Clock block.</td>
<td></td>
</tr>
<tr>
<td>3  Set the sample time of the clock signal to the sample time of the Sine Wave block.</td>
<td></td>
</tr>
</tbody>
</table>

---

### Sample-Based Mode

Sample-based mode uses this formula to compute the output of the Sine Wave block.

\[
y = A \sin(2\pi(k + o)/p) + b
\]

- **A** is the amplitude of the sine wave.
- **p** is the number of time samples per sine wave period.
- **k** is a repeating integer value that ranges from 0 to **p-1**.
- **o** is the offset (phase shift) of the signal.
- **b** is the signal bias.

In this mode, Simulink sets **k** equal to 0 at the first time step and computes the block output, using the formula. At the next time step, Simulink increments **k** and recomputes the output of the block. When **k** reaches **p**, Simulink resets **k** to 0 before computing the block output. This process continues until the end of the simulation.

The sample-based method of computing block output at a given time step does not depend on the output of the previous time steps. Therefore, this mode avoids the
accumulation of round-off errors. Sample-based mode supports reset semantics in subsystems that offer it. For example, if a Sine Wave block is in a resettable subsystem that receives a reset trigger, the repeating integer \( k \) resets and the block output resets to its initial condition.

### Ports

#### Output

**Port_1 — Sine wave output signal**

*scalar | vector*

Output sine wave signal created based on the block parameter values.

**Data Types:** double

### Parameters

**Sine type — Type of sine wave**

*Time based (default) | Sample based*

Specify the type of sine wave that this block generates. Some parameters in the dialog box appear depending on whether you select time-based or sample-based.

**Programmatic Use**

**Block Parameter:** `SineType`

**Type:** character vector

**Values:** `'Time based' | 'Sample based'`

**Default:** `'Time based'`

**Time \((t)\) — Source of time variable**

*Use simulation time (default) | Use external signal*

Specify whether to use simulation time as the source of values for the time variable, or an external source. If you specify an external time source, the block creates an input port for the time source. When you select an external time source, the block is the same as the Sine Wave Function block.
Programmatic Use
Block Parameter: TimeSource
Type: character vector
Values: 'Use simulation time' | 'Use external signal'
Default: 'Use simulation time'

Amplitude — Amplitude of the sine wave
1 (default) | scalar | vector
Specify the amplitude of the output sine wave signal.

Programmatic Use
Block Parameter: Amplitude
Type: character vector
Value: scalar
Default: '1'

Bias — Constant added to sine wave
0 (default) | scalar | vector
Specify the constant value added to the sine to produce the output.

Programmatic Use
Block Parameter: Bias
Type: character vector
Value: scalar
Default: '0'

Frequency (rad/sec) — Frequency of sine wave
1 (default) | scalar | vector
Specify the frequency, in rad/sec.

Dependencies
To enable this parameter, set Sine type to Time based.

Programmatic Use
Block Parameter: Frequence
Type: character vector
Value: scalar
Default: '1'
**Phase (rad) — Phase shift of sine wave**

θ (default) | scalar | vector

Specify the phase shift of the sine wave.

You cannot configure this parameter to appear in the generated code as a tunable global variable if you set **Time (t)** to Use simulation time. For example, if you set **Default parameter behavior** to Tunable or apply a storage class to a Simulink.Parameter object, the Phase parameter does not appear in the generated code as a tunable global variable.

To generate code so that you can tune the phase during execution, set **Time (t)** to Use external signal. You can provide your own time input signal or use a Digital Clock block to generate the time signal. For an example, see “Tune Phase Parameter of Sine Wave Block During Code Execution” (Simulink Coder).

**Dependencies**

To enable this parameter, set **Sine type** to Time based.

**Programmatic Use**

**Block Parameter:** Phase  
**Type:** character vector  
**Value:** scalar  
**Default:** '0'

**Samples per period — Samples per period**

θ (default) | integer scalar | integer vector

Specify the number of samples per period.

**Dependencies**

To enable this parameter, set **Sine type** to Sample based.

**Programmatic Use**

**Block Parameter:** Samples  
**Type:** character vector  
**Value:** scalar  
**Default:** '10'

**Number of offset samples — Offset in number of time samples**

θ (default) | integer scalar | integer vector
Specify the offset (discrete phase shift) in number of sample times.

**Dependencies**

To enable this parameter, set **Sine type** to **Sample based**.

**Programmatic Use**

**Block Parameter**: Offset
- **Type**: character vector
- **Value**: scalar
- **Default**: '0'

**Sample time — Sample period**

0 (default) | scalar | vector

Specify the sample period in seconds. The default is 0. If the sine type is sample-based, the sample time must be greater than 0. See “Specify Sample Time”.

**Programmatic Use**

**Block Parameter**: SampleTime
- **Type**: character vector
- **Value**: scalar
- **Default**: '0'

**Interpret vector parameters as 1-D — Output dimensions for one-row or one-column matrices**

off (default) | on

Specify the output dimensions to be a 1-D vector signal when other parameters are one-row and one-column matrices. If you do not select this box, the block outputs a signal of the same dimensionality as the numeric parameters. See “Determining the Output Dimensions of Source Blocks” in the Simulink documentation. This parameter is not available when an external signal specifies time. In this case, if numeric parameters are column or row matrix values, the output is a 1-D vector.

**Programmatic Use**

**Block Parameter**: VectorParams1D
- **Type**: character vector
- **Values**: 'off' | 'on'
- **Default**: 'on'
Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>double</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>yes</td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>no</td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
</tr>
</tbody>
</table>

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

Depends on absolute time when placed inside a triggered subsystem hierarchy. These blocks do not reference absolute time when configured for sample-based operation. In time-based operation, they depend on absolute time.

See Also
Sine Wave Function | Sine, Cosine

Introduced before R2006a
Sine Wave Function

Generate sine wave, using external signal as time source

Library: Simulink / Math Operations

Description

The Sine Wave Function block outputs a sinusoidal waveform. The block can operate in time-based or sample-based mode.

Note This block is the same as the Sine Wave block that appears in the Math Operations library. If you select Use simulation time for the Time parameter in the block dialog box, you get the Sine Wave Function block.

Time-Based Mode

The block calculates the output waveform.

\[ y = amplitude \times \sin(frequency \times time + phase) + bias. \]

In time-based mode, the value of the Sample time parameter determines whether the block operates in continuous mode or discrete mode.

- 0 (the default) causes the block to operate in continuous mode.
- >0 causes the block to operate in discrete mode.

For more information, see “Specify Sample Time”.
Block Behavior in Continuous Mode

When operating in continuous mode, the Sine Wave block can become inaccurate due to loss of precision as time becomes very large.

Block Behavior in Discrete Mode

A **Sample time** parameter value greater than zero causes the block to behave as if it were driving a Zero-Order Hold block whose sample time is set to that value.

This way, you can build models with sine wave sources that are purely discrete, rather than models that are hybrid continuous/discrete systems. Hybrid systems are inherently more complex and as a result take more time to simulate.

In discrete mode, this block uses a differential incremental algorithm instead of one based on absolute time. As a result, the block can be useful in models intended to run for an indefinite length of time, such as in vibration or fatigue testing.

The differential incremental algorithm computes the sine based on the value computed at the previous sample time. This method uses the following trigonometric identities:

\[
\begin{align*}
\sin(t + \Delta t) &= \sin(t) \cos(\Delta t) + \sin(\Delta t) \cos(t) \\
\cos(t + \Delta t) &= \cos(t) \cos(\Delta t) - \sin(t) \sin(\Delta t)
\end{align*}
\]

In matrix form, these identities are:

\[
\begin{bmatrix}
\sin(t + \Delta t) \\
\cos(t + \Delta t)
\end{bmatrix} = 
\begin{bmatrix}
\cos(\Delta t) & \sin(\Delta t) \\
-\sin(\Delta t) & \cos(\Delta t)
\end{bmatrix} 
\begin{bmatrix}
\sin(t) \\
\cos(t)
\end{bmatrix}
\]

Because \(\Delta t\) is constant, the following expression is a constant:

\[
\begin{bmatrix}
\cos(\Delta t) & \sin(\Delta t) \\
-\sin(\Delta t) & \cos(\Delta t)
\end{bmatrix}
\]

Therefore, the problem becomes one of a matrix multiplication of the value of \(\sin(t)\) by a constant matrix to obtain \(\sin(t + \Delta t)\).

Discrete mode reduces but does not eliminate the accumulation of round-off errors, for example, \((4*\text{eps})\). This accumulation can happen because computation of the block output at each time step depends on the value of the output at the previous time step.
Methods to Handle Round-Off Errors in Discrete Mode

To handle round-off errors when the Sine Wave block operates in time-based discrete mode, use one of these methods.

<table>
<thead>
<tr>
<th>Method</th>
<th>Rationale</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert a Saturation block directly downstream of the Sine Wave block.</td>
<td>By setting saturation limits on the Sine Wave block output, you can remove overshoot due to accumulation of round-off errors.</td>
</tr>
<tr>
<td>Set up the Sine Wave block to use the sin() math library function to calculate block output.</td>
<td>The sin() math library function computes block output at each time step independently of output values from other time steps, preventing the accumulation of round-off errors.</td>
</tr>
</tbody>
</table>

1. On the Sine Wave block dialog box, set **Time** to **Use external signal** so that an input port appears on the block icon.
2. Connect a clock signal to this input port using a Digital Clock block.
3. Set the sample time of the clock signal to the sample time of the Sine Wave block.

Sample-Based Mode

Sample-based mode uses this formula to compute the output of the Sine Wave block.

\[ y = A \sin(2\pi(k + o)/p) + b \]

- \( A \) is the amplitude of the sine wave.
- \( p \) is the number of time samples per sine wave period.
- \( k \) is a repeating integer value that ranges from 0 to \( p-1 \).
- \( o \) is the offset (phase shift) of the signal.
- \( b \) is the signal bias.

In this mode, Simulink sets \( k \) equal to \( 0 \) at the first time step and computes the block output, using the formula. At the next time step, Simulink increments \( k \) and recomputes
the output of the block. When \( k \) reaches \( p \), Simulink resets \( k \) to 0 before computing the block output. This process continues until the end of the simulation.

The sample-based method of computing block output at a given time step does not depend on the output of the previous time steps. Therefore, this mode avoids the accumulation of round-off errors. Sample-based mode supports reset semantics in subsystems that offer it. For example, if a Sine Wave block is in a resettable subsystem that receives a reset trigger, the repeating integer \( k \) resets and the block output resets to its initial condition.

**Ports**

**Input**

**Port_1 — Time source signal**
scalar

Input signal representing the time source in the sine wave calculation.
Data Types: double

**Output**

**Output 1 — Output sine wave signal**
scalar

Output signal that is the created sine wave.
Data Types: double

**Parameters**

**Sine type — Type of sine wave**
Time based (default) | Sample based

Specify the type of sine wave that this block generates. Some parameters in the dialog box appear depending on whether you select time-based or sample-based.
**Programmatic Use**
**Block Parameter:** SineType
**Type:** character vector
**Values:** 'Time based' | 'Sample based'
**Default:** 'Time based'

**Time (t) — Source of time variable**
*Use external signal (default) | Use simulation time*

Specify whether to use simulation time as the source of values for the time variable, or an external source. If you specify an external time source, the block creates an input port for the time source.

**Programmatic Use**
**Block Parameter:** TimeSource
**Type:** character vector
**Values:** 'Use simulation time' | 'Use external signal'
**Default:** 'Use external signal'

**Amplitude — Amplitude of the sine wave**
1 (default) | scalar

Specify the amplitude of the output sine wave signal.

**Programmatic Use**
**Block Parameter:** Amplitude
**Type:** character vector
**Value:** scalar
**Default:** '1'

**Bias — Constant added to sine wave**
0 (default) | scalar

Specify the constant value added to the sine to produce the output.

**Programmatic Use**
**Block Parameter:** Bias
**Type:** character vector
**Value:** scalar
**Default:** '0'

**Frequency (rad/sec) — Frequency of sine wave**
1 (default) | scalar
Specify the frequency, in radians per second.

**Dependency**

To enable this parameter, set **Sine type** to **Time based**.

**Programmatic Use**

**Block Parameter:** Frequency  
**Type:** character vector  
**Value:** scalar  
**Default:** '1'

**Phase (rad) — Phase shift of sine wave**  
0 (default) | scalar

Specify the phase shift of the sine wave.

You cannot configure this parameter to appear in the generated code as a tunable global variable if you set **Time (t)** to **Use simulation time**. For example, if you set **Default parameter behavior** to **Tunable** or apply a storage class to a Simulink.Parameter object, the **Phase** parameter does not appear in the generated code as a tunable global variable.

To generate code so that you can tune the phase during execution, set **Time (t)** to **Use external signal**. You can provide your own time input signal or use a Digital Clock block to generate the time signal. For an example, see “Tune Phase Parameter of Sine Wave Block During Code Execution” (Simulink Coder).

**Dependencies**

To enable this parameter, set **Sine type** to **Time based**.

**Programmatic Use**

**Block Parameter:** Phase  
**Type:** character vector  
**Value:** scalar  
**Default:** '0'

**Samples per period — Samples per period**  
0 (default) | integer

Specify the number of samples per period.
**Dependencies**

To enable this parameter, set **Sine type** to **Sample based**.

**Programmatic Use**

**Block Parameter**: Samples  
**Type**: character vector  
**Value**: scalar  
**Default**: '10'

**Number of offset samples** — **Offset in number of time samples**

0 (default) | integer

Specify the offset (discrete phase shift) in number of sample times.

**Dependencies**

To enable this parameter, set **Sine type** to **Sample based**.

**Programmatic Use**

**Block Parameter**: Offset  
**Type**: character vector  
**Value**: scalar  
**Default**: '0'

**Sample time** — **Sample period**

0 (default) | scalar

Specify the sample period in seconds. The default is 0. If the sine type is sample-based, the sample time must be greater than 0. See “Specify Sample Time”.

**Programmatic Use**

**Block Parameter**: SampleTime  
**Type**: character vector  
**Value**: scalar  
**Default**: '0'

**Interpret vector parameters as 1-D** — **Output dimensions for one-row or one-column matrices**

off (default) | on

Specify the output dimensions to be a 1-D vector signal when other parameters are one-row and one-column matrices. If you do not select this box, the block outputs a signal of the same dimensionality as the numeric parameters. See “Determining the Output
Dimensions of Source Blocks” in the Simulink documentation. This parameter is not available when an external signal specifies time. In this case, if numeric parameters are column or row matrix values, the output is a 1-D vector.

**Programmatic Use**

**Block Parameter:** VectorParams1D  
**Type:** character vector  
**Values:** 'off' | 'on'  
**Default:** 'on'

## Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>double</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>yes</td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>no</td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
</tr>
</tbody>
</table>

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Depending on absolute time when placed inside a triggered subsystem hierarchy. These blocks do not reference absolute time when configured for sample-based operation. In time-based operation, they depend on absolute time.
See Also
Sine Wave | Sine, Cosine

Introduced before R2006a
Slider

Tune parameter value with sliding scale

Library: Simulink / Dashboard

Description

The Slider block tunes the value of the connected block parameter during simulation. For example, you can connect the Slider block to a Gain block in your model and adjust its value during simulation. You can modify the range of the Slider block's scale to fit your data. Use the Slider block with other Dashboard blocks to create an interactive dashboard to control your model.

Connecting Dashboard Blocks

Dashboard blocks do not use ports to make connections. To connect Dashboard blocks to variables and block parameters in your model, use connect mode. Connect mode facilitates the process of connecting Dashboard blocks in your model, especially when you want to connect multiple blocks at once. If you only want to connect a single Dashboard block, you can also use the Connection table in the block dialog box to make the connection.

Tip To facilitate understanding and debugging your model, you can modify Dashboard block connections in your model during normal and accelerator mode simulations.

Note Dashboard blocks cannot connect to variables until you update your model diagram. To connect Dashboard blocks to variables or modify variable values between opening your model and running a simulation, update your model diagram using Ctrl+D.

To enter connect mode, click the Connect button that appears above your unconnected Dashboard block when you pause on it.
In connect mode, when you select one or more signals or blocks, a list of parameters and signals available for connection appears. Select a signal or parameter from the list to connect the selected Dashboard block. To connect another Dashboard block, pause on the block and click the **Connect** button above it. Then, make a selection of signals and blocks in your model, and choose a signal or parameter to connect.
When you finish connecting the Dashboard blocks in your model, click the **Exit** button in the upper-right of the canvas to exit connect mode.

**Tip** You can hide the message shown on unconnected blocks using the `set_param` function with the `ShowInitialText` block parameter.

**Parameter Logging**

Tunable parameters connected to Dashboard blocks are logged to the Simulation Data Inspector, where you can view parameter values along with logged signal data. You can access logged parameter data in the MATLAB workspace by exporting the parameter data from the Simulation Data Inspector UI or by using the `Simulink.sdi.exportRun` function. For more information about exporting data with the Simulation Data Inspector UI, see “Export Data from the Simulation Data Inspector”. The parameter data is stored in a `Simulink.SimulationData.Parameter` object, accessible as an element in the exported `Simulink.SimulationData.Dataset`.

1-1901
Limitations

- Dashboard blocks can only connect to real scalar signals.
- You cannot use the **Connection** table to connect a Dashboard block to a block that is commented out. When you connect a Dashboard block to a commented block using connect mode, the Dashboard block does not display the connected value until you uncomment the block.
- Dashboard blocks cannot connect to signals inside referenced models.
- When you simulate a model hierarchy, Dashboard blocks inside referenced models do not update.
- Dashboard blocks do not support rapid accelerator simulation.
- Some signals are not available during accelerator mode simulations. To view such signals on a Dashboard block, mark the signal for logging.
- Parameters specified by indexing a variable array do not appear in the **Connection** table. For example, a block parameter defined as `engine(1)` using the variable `engine` does not appear in the table.

To access the parameter in the **Connection** table, assign the indexed value to a scalar variable, such as `engine_1`. Then, use the scalar variable to define the block parameter.

- When you connect a Dashboard block to a variable or parameter during simulation, the data for that variable or parameter does not get logged to the Simulation Data Inspector. To log variable and parameter data to the Simulation Data Inspector, connect the Dashboard block to the variable or parameter prior to simulation.
- When you simulate a model in external mode with the **Default parameter behavior** set to **Inlined**, Dashboard blocks can appear to change parameter and variable values. However, the change does not propagate to the simulation. For example, the Gain block displays changes made to its **Gain** parameter using the Dashboard blocks, but the **Gain** value used in the simulation does not change.

Parameters

**Connection** — Select a variable or block parameter to connect
variable and parameter connection options

Select the variable or block parameter to control using the **Connection** table. Populate the **Connection** table by selecting one or more blocks in your model. Select the radio
button next to the variable or parameter you want to control, and click **Apply**. To facilitate understanding and debugging your model, you can connect Dashboard blocks to variables and parameters in your model during simulation.

**Note** To see workspace variables in the connection table, update the model diagram using **Ctrl+D**.

**Programmatic Use**

To programmatically connect a Dashboard block to a tunable parameter or a variable, use a `Simulink.HMI.ParamSourceInfo` object. The `Simulink.HMI.ParamSourceInfo` object contains four properties. Some of the properties apply to connecting Dashboard blocks to parameters, and some apply to connecting Dashboard blocks to variables. Not all fields have a value for a connection because a given Dashboard block connects to either a parameter or a variable.

**Block Parameter:** Binding  
**Type:** `Simulink.HMI.ParamSourceInfo`  
**Default:** `[]`  

**Scale Type — Type of scale**

**Linear** (default) | **Log**

Type of scale displayed on the control. **Linear** specifies a linear scale, and **Log** specifies a logarithmic scale.

**Programmatic Use**

**Block Parameter:** `ScaleType`  
**Type:** string or character vector  
**Values:** 'Linear' | 'Log'  
**Default:** 'Linear'

**Minimum — Minimum tick mark value**

0 (default) | scalar

Finite, real, double, scalar value specifying the minimum tick mark value for the arc. The minimum must be less than the value entered for the maximum.

**Programmatic Use**

To programmatically set the `Minimum` parameter, use a 1-by-3 vector containing values for the `Minimum`, `Tick Interval`, and `Maximum` parameters, in that order. To use the
auto value for the Tick Interval, leave the Tick Interval position in the vector empty or specify -1.

**Block Parameter:** Limits
**Type:** 1x3 vector
**Default:** [0 -1 100]

**Maximum — Maximum tick mark value**
100 (default) | scalar

Finite, real, double, scalar value specifying the maximum tick mark value for the arc. The maximum must be greater than the value entered for the minimum.

**Programmatic Use**

To programmatically set the Maximum parameter, use a 1-by-3 vector containing values for the Minimum, Tick Interval, and Maximum parameters, in that order. To use the auto value for the Tick Interval, leave the Tick Interval position in the vector empty or specify -1.

**Block Parameter:** Limits
**Type:** 1x3 vector
**Default:** [0 -1 100]

**Tick Interval — Interval between major tick marks**

auto (default) | scalar

Finite, real, positive, integer, scalar value specifying the interval of major tick marks on the arc. When set to auto, the block automatically adjusts the tick interval based on the minimum and maximum values.

**Programmatic Use**

To programmatically set the Tick Interval parameter, use a 1-by-3 vector containing values for the Minimum, Tick Interval, and Maximum parameters, in that order. To use the auto value for the Tick Interval, leave the Tick Interval position in the vector empty or specify -1.

**Block Parameter:** Limits
**Type:** 1x3 vector
**Default:** [0 -1 100]

**Label — Block label position**
Top (default) | Bottom | Hide
Position of the block label. When the block is connected to an element in the model, the label is the name of the connected element.

**Programmatic Use**

**Block Parameter:** LabelPosition  
**Type:** character vector  
**Values:** 'Top' | 'Bottom' | 'Hide'  
**Default:** 'Top'

---

### Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>double</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>no</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

### Extended Capabilities

**C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

Dashboard blocks are ignored for code generation.

**See Also**

Knob | Rotary Switch

---

1-1905
Topics
“Tune and Visualize Your Model with Dashboard Blocks”
“Decide How to Visualize Simulation Data”

Introduced in R2015b
Slider Gain

Vary scalar gain using slider

Library: Simulink / Math Operations

Description

The Slider Gain block performs a scalar gain that you can modify during simulation. Modify the gain using the slider parameter.

Ports

Input

Port_1 — Input signal
scalar | vector | matrix

The Slider Gain block accepts real or complex-valued scalar, vector, or matrix input. The block supports fixed-point data types.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point

Output

Port_1 — Input multiplied by gain
scalar | vector | matrix

The Slider Gain block outputs the input multiplied by a constant gain value. When the input to the block is real and gain is complex, the output is complex.
Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point

Parameters

Slider gain — Gain value
0 (default) | real value

Chose the gain value applied to the input.

Programmatic Use
Block Parameter: gain
Type: character vector
Values: real scalar
Default: '1'

Low — Lower limit of slider range
0 (default) | real value

Specify the lower limit of the slider range.

Programmatic Use
Block Parameter: low
Type: character vector
Values: real scalar
Default: '0'

High — Upper limit of slider range
2 (default) | real value

Specify the upper limit of the slider range.

Programmatic Use
Block Parameter: high
Type: character vector
Values: real scalar
Default: '2'
Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>double</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

PLC Code Generation
Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

See Also
Gain

Introduced before R2006a
Spectrum Analyzer

Display frequency spectrum

**Library:**
- DSP System Toolbox / Sinks
- DSP System Toolbox HDL Support / Sinks

**Description**

The Spectrum Analyzer block, referred to here as the scope, displays the frequency spectra of signals.
You can use the Spectrum Analyzer block in models running in Normal or Accelerator simulation modes. You can also use the Spectrum Analyzer block in models running in Rapid Accelerator or External simulation modes, with some limitations.

You can use the Spectrum Analyzer block inside all subsystems and conditional subsystems. Conditional subsystems include enabled subsystems, triggered subsystems, enabled and triggered subsystems, and function-call subsystems. See “Conditionally Executed Subsystems Overview” for more information.

**Measurements**

- **Cursors (DSP System Toolbox)** — Measure signal values using vertical and horizontal cursors.
- **Peak Finder (DSP System Toolbox)** — Find maxima, showing the x-axis values at which they occur.
- **Channel Measurements (DSP System Toolbox)** — Measure the occupied bandwidth or adjacent channel power ratio (ACPR).
- **Distortion Measurements (DSP System Toolbox)** — Measure harmonic distortion and intermodulation distortion.
- **CCDF Measurements (DSP System Toolbox)** — Measure the complimentary cumulative distribution function. CCDF measurements show the probability of a signal’s instantaneous power being a specified level above the signal’s average power.
- **“Spectral Masks” (DSP System Toolbox)** — Visualize spectrum limits and compare spectrum values to specification values.

**Programmatic Control**

You can configure and display Spectrum Analyzer settings from the command line with the `spbscopes.SpectrumAnalyzerConfiguration` object.

**Ports**

**Input**

**Port_1 — Signals to visualize**

scalar | vector | matrix | array
Connect the signals you want to visualize. You can have up to 96 input ports. Input signals can have these characteristics:

- **Signal Domain** — Frequency or time signals
- **Type** — Discrete (sample-based and frame-based).
- **Data type** — Any data type that Simulink supports. See “Data Types Supported by Simulink”.
- **Dimension** — One dimensional (vector), two dimensional (matrix), or multidimensional (array). Input must have fixed number of channels. See “Signal Dimensions” and “Determine Output Signal Dimensions”.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

## Parameters

### Spectrum Settings

The **Spectrum Settings** pane appears at the right side of the Spectrum Analyzer window. These settings control how the spectrum is calculated. To show the Spectrum Settings, in the Spectrum Analyzer menu, select **View > Spectrum Settings** or use the button in the toolbar.

**Main options**

**Input domain — Domain of the input signal**

**Time** (default) | **Frequency**

The domain of the input signal you want to visualize. If you visualize time-domain signals, the signal is transformed to the frequency spectrum based on the algorithm specified by the **Method** parameter.

**Programmatic Use**

See **InputDomain**.

**Type — Type of spectrum to display**

**Power** (default) | **Power density** | **RMS**
Power — Spectrum Analyzer shows the power spectrum.

Power density — Spectrum Analyzer shows the power spectral density. The power spectral density is the magnitude of the spectrum normalized to a bandwidth of 1 hertz.

RMS — Spectrum Analyzer shows the root mean squared spectrum.

Tunable: Yes

Programmatic Use

See .

View — Spectrum view
Spectrum (default) | Spectrogram | Spectrum and spectrogram

Spectrum — Spectrum Analyzer shows the spectrum.

Spectrogram — Spectrum Analyzer shows the spectrogram, which displays frequency content over time. The most recent spectrogram update is at the bottom of the display, and time scrolls from the bottom to the top of the display.

Spectrum and spectrogram — Spectrum Analyzer shows both the spectrum and spectrogram.

Tunable: Yes

Programmatic Use

See ViewType.

Sample rate — Sample rate of the input signal in hertz
Inherited (default) | positive scalar

Select Inherited to use the same sample rate as the input signal. To specify a sample rate, delete Inherited and enter a sample rate value.

Programmatic Use

See SampleRate.

Method — Spectrum estimation method
Welch (default) | Filter Bank
Select Welch or Filter Bank as the spectrum estimation method. For more details about the two spectrum estimation algorithms, see “Algorithms” (DSP System Toolbox).

**Tunable:** No

**Dependency**

To use this parameter, set “Input domain” (DSP System Toolbox) to Time.

**Programmatic Use**

See `Method`.

**Full frequency span — Use entire Nyquist frequency interval**

`on (default) | off`

Select this check box to compute and plot the spectrum over the entire “Nyquist frequency interval” (DSP System Toolbox).

**Tunable:** Yes

**Dependency**

To use this parameter, set “Input domain” (DSP System Toolbox) to Time.

**Programmatic Use**

See `FrequencySpan`.

**Span (Hz) — Frequency span in hertz**

`10e3 (default) | real positive scalar`

Specify the frequency span in hertz. Use this parameter with the CF (Hz) parameter to define the frequency span around a center frequency. This parameter defines the range of values shown on the Frequency axis in the Spectrum Analyzer window.

**Tunable:** Yes

**Dependencies**

To use this parameter, you must:

- Set “Input domain” (DSP System Toolbox) to Time.
- Clear the “Full frequency span” (DSP System Toolbox) check box.
• Set the **Span (Hz)/Fstart (Hz)** dropdown to **Span (Hz)**.

**Programmatic Use**

See FrequencySpan and Span.

**CF (Hz) — Center frequency in hertz**

0 (default) | scalar

Specify the center frequency, in hertz. Use this parameter with the “Span (Hz)” (DSP System Toolbox) parameter to define the frequency span around a center frequency. This parameter defines the value shown at the middle point of the Frequency axis on the Spectrum Analyzer window.

**Tunable**: Yes

**Dependencies**

To use this parameter, you must:

• Set “Input domain” (DSP System Toolbox) to **Time**.
• Clear the “Full frequency span” (DSP System Toolbox) check box.
• Set the **Span (Hz)/Fstart (Hz)** dropdown to “Span (Hz)” (DSP System Toolbox).

**Programmatic Use**

See CenterFrequency.

**FStart (Hz) — Start frequency in hertz**

-5e3 (default) | scalar

Specify the start frequency in hertz. Use this parameter with the “FStop (Hz)” (DSP System Toolbox) parameter to define the range of frequency-axis values using start frequency and stop frequency. This parameter defines the value shown at the leftmost side of the Frequency axis on the Spectrum Analyzer window.

**Tunable**: Yes

**Dependencies**

To use this parameter, you must:

• Set “Input domain” (DSP System Toolbox) to **Time**.
• Clear the “Full frequency span” (DSP System Toolbox) check box.
• Set the Span (Hz)/FStart (Hz) dropdown to FStart (Hz).

**Programmatic Use**

See StartFrequency.

**FStop (Hz) — Stop frequency in hertz**

5e3 (default) | scalar

Specify the stop frequency, in hertz. Use this parameter with the “FStart (Hz)” (DSP System Toolbox) parameter to define the range of Frequency axis values. This parameter defines the value shown at the rightmost side of the Frequency axis on the Spectrum Analyzer window.

**Tunable:** Yes

**Dependencies**

To use this parameter, you must:

• Set “Input domain” (DSP System Toolbox) to Time.
• Clear the “Full frequency span” (DSP System Toolbox) check box.
• Set the Span (Hz)/FStart (Hz) dropdown to “FStart (Hz)” (DSP System Toolbox).

**Programmatic Use**

See StopFrequency.

**Frequency (Hz) — Frequency vector**

Auto (default) | Input port | monotonically increasing vector

Set the frequency vector which determines the x-axis of the display.

• Auto — The frequency vector is calculated from the length of the input. See “Frequency Vector” (DSP System Toolbox).
• Input port — When selected, an input port appears on the block for the frequency vector input.
• Custom vector — Enter a custom vector as the frequency vector. The length of the custom vector must be equal to the frame size of the input signal.

**Tunable:** No
Dependency

To use this parameter, set “Input domain” (DSP System Toolbox) to Frequency.

Programmatic Use

See FrequencyVector.

RBW (Hz) — Resolution bandwidth
Auto (default) | Input port | positive scalar

The resolution bandwidth in hertz. This parameter defines the smallest positive frequency that can be resolved. By default, this parameter is set to Auto. In this case, the Spectrum Analyzer determines the appropriate value to ensure that there are 1024 RBW intervals over the specified frequency span.

If you set this parameter to a numeric value, the value must allow at least two RBW intervals over the specified frequency span. In other words, the ratio of the overall frequency span to RBW must be greater than two:

\[
\frac{\text{span}}{\text{RBW}} > 2
\]

For frequency input only, you can use an input port to set the RBW value.

Tunable: Yes

Dependency

To use this parameter, set either:

- “Input domain” (DSP System Toolbox) to Time and the RBW (Hz)/Window length/Number of frequency bands dropdown to RBW (Hz).
- “Input domain” (DSP System Toolbox) to Frequency.

Programmatic Use

See RBW.

Input units — Units of frequency input
Auto (default) | dBm | dBV | dBW | Vrms | Watts
Select the units of the frequency-domain input. This property allows the Spectrum Analyzer to scale frequency data if you choose a different display unit with the “Units” (DSP System Toolbox) property.

**Tunable:** No

**Dependency**

This option is only available when “Input domain” (DSP System Toolbox) is set to Frequency.

**Programmatic Use**

See InputUnits.

**Window length — Length of window in samples**

1024 (default) | integer greater than 2

The length of the window, in samples. The window length used to control the frequency resolution and compute the spectral estimates. The window length must be an integer greater than 2.

**Dependencies**

To use this parameter, set:

- “Method” (DSP System Toolbox) to Welch
- Set the RBW (Hz)/Window length/Number of frequency bands dropdown to Window Length

**Dependency**

To use this parameter, set “Input domain” (DSP System Toolbox) to Time.

**Programmatic Use**

See WindowLength.

**Number of frequency bands — FFT length**

Auto (default) | positive integer

Specify the fast Fourier transform (FFT) length to control the number of frequency bands. If the value is Auto, the Spectrum Analyzer uses the entire frame size to estimate the spectrum. If you specify the number of frequency bands, you set the input buffer size.
Dependencies

To use this parameter, set:

• “Method” (DSP System Toolbox) to Filter Bank
• Set the RBW (Hz)/Window length/Number of frequency bands dropdown to Number of frequency bands

Programmatic Use

See FFTLength

Taps per band — Number of filter taps

12 (default) | positive even integer

Specify the number of filter taps or coefficients for each frequency band. This number must be a positive even integer. This value corresponds to the number of filter coefficients per polyphase branch. The total number of filter coefficients is equal to Taps Per Band + FFT Length.

Dependency

To use this parameter, you must set the RBW (Hz)/Window length/Number of frequency bands dropdown to “Number of frequency bands” (DSP System Toolbox).

Programmatic Use

See NumTapsPerBand.

NFFT — Number of FFT points

Auto (default) | positive integer

Specify the length of the FFT that Spectrum Analyzer uses to compute spectral estimates. Acceptable options are Auto or a positive integer.

The NFFT value must be greater than or equal to the value of the Window length parameter. By default, when NFFT is set to Auto, the Spectrum Analyzer sets NFFT equal to the value of Window length. When in RBW mode, the specified RBW value is used to calculate an FFT length that equals the window length.

When this parameter is set to a positive integer, this parameter is equivalent to the n parameter of the fft function.
Dependencies

To use this parameter, you must set the **RBW (Hz)/Window length/Number of frequency bands** dropdown to “Window length” (DSP System Toolbox).

**Programmatic Use**

See FFTLength.

**Samples/update — Required number of input samples**

This property is read-only.

The number of input samples required to compute one spectral update. You cannot modify this parameter; it is shown in the spectrum analyzer for informational purposes only. This parameter is directly related to **RBW (Hz)/Window length/Number of frequency bands**. For more details, see “Algorithms” (DSP System Toolbox).

If the input does not have enough samples to achieve the resolution bandwidth that you specify, Spectrum Analyzer produces a message on the display.

---

**Spectrogram Settings**

**Channel — Spectrogram channel**

Select the signal channel for which the spectrogram settings apply.

**Dependencies**

To use this option, set “View” (DSP System Toolbox) to **Spectrogram** or **Spectrum** and **spectrogram**.

**Programmatic Use**

See SpectrogramChannel.
**Time res. (s) — Time resolution in seconds**
Auto (default) | positive number

Time resolution is the amount of data, in seconds, used to compute a spectrogram line. The minimum attainable resolution is the amount of time it takes to compute a single spectral estimate. The tooltip displays the minimum attainable resolution given the current settings.

The time resolution value is determined based on frequency resolution method, the RBW setting, and the time resolution setting.

<table>
<thead>
<tr>
<th>Method</th>
<th>Frequency Resolution Method</th>
<th>Frequency Resolution Setting</th>
<th>Time Resolution Setting</th>
<th>Resulting Time Resolution in Seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>Welch or Filter Bank</td>
<td>RBW (Hz)</td>
<td>Auto</td>
<td>Auto</td>
<td>1/RBW</td>
</tr>
<tr>
<td>Welch or Filter Bank</td>
<td>RBW (Hz)</td>
<td>Auto</td>
<td>Manually entered</td>
<td>Time Resolution</td>
</tr>
<tr>
<td>Welch or Filter Bank</td>
<td>RBW (Hz)</td>
<td>Manually entered</td>
<td>Auto</td>
<td>1/RBW</td>
</tr>
<tr>
<td>Welch or Filter Bank</td>
<td>RBW (Hz)</td>
<td>Manually entered</td>
<td>Manually entered</td>
<td>Must be equal to or greater than the minimum attainable time resolution, 1/RBW. Several spectral estimates are combined into one spectrogram line to obtain the desired time resolution. Interpolation is used to obtain time resolution values that are not integer multiples of 1/RBW.</td>
</tr>
<tr>
<td>Welch</td>
<td>Window length</td>
<td>—</td>
<td>Auto</td>
<td>1/RBW</td>
</tr>
<tr>
<td>Method</td>
<td>Frequency Resolution Method</td>
<td>Frequency Resolution Setting</td>
<td>Time Resolution Setting</td>
<td>Resulting Time Resolution in Seconds</td>
</tr>
<tr>
<td>-----------------</td>
<td>-----------------------------</td>
<td>------------------------------</td>
<td>-------------------------</td>
<td>--------------------------------------</td>
</tr>
<tr>
<td>Welch</td>
<td>Window length</td>
<td>—</td>
<td>Manually entered</td>
<td>Must be equal to or greater than the minimum attainable time resolution. Several spectral estimates are combined into one spectrogram line to obtain the desired time resolution. Interpolation is used to obtain time resolution values that are not integer multiples of 1/RBW.</td>
</tr>
<tr>
<td>Filter Bank</td>
<td>Number of frequency bands</td>
<td>—</td>
<td>Auto</td>
<td>1/RBW</td>
</tr>
<tr>
<td>Filter Bank</td>
<td>Number of frequency bands</td>
<td>—</td>
<td>Manually entered</td>
<td>Must be equal to or greater than the minimum attainable time resolution, 1/RBW.</td>
</tr>
</tbody>
</table>

**Tunable:** Yes

**Dependency**

To use this option, set “View” (DSP System Toolbox) to Spectrogram or Spectrum and spectrogram.

**Programmatic Use**

See TimeResolution.

**Time span — Time span in seconds**

Auto (default) | positive scalar
The time span over which the Spectrum Analyzer displays the spectrogram specified in seconds. The time span is the product of the desired number of spectral lines and the time resolution. The tooltip displays the minimum allowable time span, given the current settings. If the time span is set to Auto, 100 spectral lines are used.

**Tunable:** Yes

**Dependency**

To use this option, set “View” (DSP System Toolbox) to Spectrogram or Spectrum and spectrogram.

**Programmatic Use**

See TimeSpan.

**Window Options**

**Overlap (%) — Segment overlap percentage**

0 (default) | scalar between 0 and 100

This parameter defines the amount of overlap between the previous and current buffered data segments. The overlap creates a window segment that is used to compute a spectral estimate. The value must be greater than or equal to zero and less than 100.

**Tunable:** Yes

**Programmatic Use**

See OverlapPercent.

**Window — Windowing method**

Hann (default) | Rectangular | Blackman-Harris | Chebyshev | Flat Top | Hamming | Kaiser | custom window function name

The windowing method to apply to the spectrum. Windowing is used to control the effect of sidelobes in spectral estimation. The window you specify affects the window length required to achieve a resolution bandwidth and the required number of samples per update. For more information about windowing, see “Windows” (Signal Processing Toolbox).

**Tunable:** Yes
**Programmatic Use**

See `Window`.

**Attenuation — Sidelobe attenuation**

60 (default) | scalar greater than or equal to 45

The sidelobe attenuation in decibels (dB). The value must be greater than or equal to 45.

**Dependency**

This parameter applies only when you set the `Window` parameter to Chebyshev or Kaiser.

**Programmatic Use**

See `SidelobeAttenuation`.

**NENBW — Normalized effective noise bandwidth**

scalar

This property is read-only.

The normalized effective noise bandwidth of the window. You cannot modify this parameter; it is shown for informational purposes only. This parameter is a measure of the noise performance of the window. The value is the width of a rectangular filter that accumulates the same noise power with the same peak power gain.

The rectangular window has the smallest NENBW, with a value of 1. All other windows have a larger NENBW value. For example, the Hann window has an NENBW value of approximately 1.5.

**Trace Options**

**Units — Spectrum units**

dBm (default) | dBW | Watts | Vrms | dBV | dBFS

The units of the spectrum. The available values depend on the value of the “Type” (DSP System Toolbox) parameter.

**Tunable:** Yes

**Programmatic Use**

See `SpectrumUnits`.  

1-1924
**Full scale — Full scale for dBFS units**
Auto (default) | positive real scalar

The full scale used for the decibel full scale (dBFS) units. By default, the Spectrum Analyzer uses the entire spectrum scale. Specify a positive real scalar for the dBFS full scale.

**Tunable:** Yes

**Dependencies**

To enable this parameter, set:

- “Input domain” (DSP System Toolbox) to Time
- “Units” on page 1-0 to dBFS

**Programmatic Use**

See FullScale.

**Averaging method — Smoothing method**
Running (default) | Exponential

Specify the smoothing method as:

- **Running** — Running average of the last \( n \) samples. Use the Averages property to specify \( n \).
- **Exponential** — Weighted average of samples. Use the Forgetting factor property to specify the weighted forgetting factor.

For more information about the averaging methods, see “Averaging Method” on page 1-1946.

**Programmatic Use**

See AveragingMethod.

**Averages — Number of spectral averages**
1 (default) | positive integer

Specify the number of spectral averages as a positive integer. The spectrum analyzer computes the current power spectrum estimate by computing a running average of the last \( N \) power spectrum estimates. This parameter defines the number of spectral averages, \( N \).
Dependencies

This parameter applies only when:

• “View” (DSP System Toolbox) is Spectrum or Spectrum and spectrogram.
• Averaging method is Running.

Programmatic Use

See SpectralAverages.

Forgetting factor — Weighting forgetting factor

0.9 (default) | scalar in the range (0,1]

Specify the exponential weighting as a scalar value greater than 0 and less than or equal to 1.

Dependency

This parameter applies only when the Averaging method is Exponential.

Programmatic Use

See ForgettingFactor.

Reference load — Reference load

1 (default) | positive real scalar

The reference load in ohms that the Spectrum Analyzer uses as a reference to compute power values.

Dependency

To use this parameter, set “Input domain” (DSP System Toolbox) to Time.

Programmatic Use

See ReferenceLoad.

Scale — Scale of frequency axis

Linear (default) | Logarithmic

Choose a linear or logarithm scale for the frequency axis. When the frequency span contains negative frequency values, you cannot choose the logarithmic option.
Programmatic Use

See FrequencyScale.

Offset — Constant frequency offset

\[ 0 \text{ (default)} | \text{scalar} \]

The constant frequency offset to apply to the entire spectrum, or a vector of frequencies to apply to each spectrum for multiple inputs. The offset parameter is added to the values on the Frequency axis in the Spectrum Analyzer window. This parameter is not used in any spectral computations. You must take the parameter into consideration when you set the Span (Hz) and CF (Hz) parameters to ensure that the frequency span is within the “Nyquist frequency interval” (DSP System Toolbox).

Dependency

To use this parameter, set “Input domain” (DSP System Toolbox) to Time.

Programmatic Use

See FrequencyOffset.

Normal trace — Normal trace view

\[ \text{on (default)} | \text{off} \]

When this check box is selected, the Spectrum Analyzer calculates and plots the power spectrum or power spectrum density. Spectrum Analyzer performs a smoothing operation by averaging several spectral estimates.

Dependencies

To clear this check box, you must first select either the “Max hold trace” (DSP System Toolbox) or the “Min hold trace” (DSP System Toolbox) parameter. This parameter applies only when “View” (DSP System Toolbox) is Spectrum or Spectrum and spectrogram.

Programmatic Use

See PlotNormalTrace.

Max hold trace — Maximum hold trace view

\[ \text{off (default)} | \text{on} \]

Select this check box to enable Spectrum Analyzer to plot the maximum spectral values of all the estimates obtained.
Dependency

This parameter applies only when “View” (DSP System Toolbox) is Spectrum or Spectrum and spectrogram.

Programmatic Use

See PlotMaxHoldTrace.

Min hold trace — Minimum hold trace view

off (default) | on

Select this check box to enable Spectrum Analyzer to plot the minimum spectral values of all the estimates obtained.

Dependency

This parameter applies only when “View” (DSP System Toolbox) is Spectrum or Spectrum and spectrogram.

Programmatic Use

See PlotMinHoldTrace.

Two-sided spectrum — Enable two-sided spectrum view

off (default) | on

Select this check box to enable a two-sided spectrum view. In this view, both negative and positive frequencies are shown. If you clear this check box, Spectrum Analyzer shows a one-sided spectrum with only positive frequencies. Spectrum Analyzer requires that this parameter is selected when the input signal is complex-valued.

Programmatic Use

See PlotAsTwoSidedSpectrum.

Configuration Properties

The Configuration Properties dialog box controls visual aspects of the Spectrum Analyzer. To open the Configuration Properties, in the Spectrum Analyzer menu, select View > Configuration Properties or select the button in the toolbar dropdown.
**Title — Display title**
character vector | string

Specify the display title. Enter `%<SignalLabel>` to use the signal labels in the Simulink model as the axes titles.

**Tunable:** Yes

**Programmatic Use**

See **Title**.

**Show legend — Display signal legend**
off (default) | on

Show signal legend. The names listed in the legend are the signal names from the model. For signals with multiple channels, a channel index is appended after the signal name. Continuous signals have straight lines before their names and discrete signals have step-shaped lines.

From the legend, you can control which signals are visible. This control is equivalent to changing the visibility in the **Style** parameters. In the scope legend, click a signal name to hide the signal in the scope. To show the signal, click the signal name again. To show only one signal, right-click the signal name, which hides all other signals. To show all signals, press **ESC**.

**Note** The legend only shows the first 20 signals. Any additional signals cannot be viewed or controlled from the legend.

**Dependency**

To enable this parameter, set “View” (DSP System Toolbox) to **Spectrum** or **Spectrum** and **spectrogram**.

**Programmatic Use**

See **ShowLegend**.

**Show grid — Show internal grid lines**
on (default) | off

Show internal grid lines on the Spectrum Analyzer
Programmatic Use

See ShowGrid.

**Y-limits (minimum) — Y-axis minimum**

-80 (default) | scalar

Specify the minimum value of the y-axis.

Programmatic Use

See YLimits.

**Y-limits (maximum) — Y-axis maximum**

20 (default) | scalar

Specify the maximum value of the y-axis.

Programmatic Use

See YLimits.

**Y-label — Y-axis label**

character vector | string

To display signal units, add (%<SignalUnits>) to the label. At the beginning of a simulation, Simulink replaces (%SignalUnits) with the units associated with the signals. For example, if you have a signal for velocity with units of m/s enter

Velocity (%<SignalUnits>)

Programmatic Use

See YLabel.

**Color map — Spectrogram colormap**

jet(256) (default) | hot(256) | bone(256) | cool(256) | copper(256) | gray(256) | parula(256) | 3-column matrix

Select the colormap for the spectrogram, or enter a three-column matrix expression for the colormap. For more information about colormaps, see colormap.

**Tunable:** Yes
Dependency

To use this parameter, set “View” (DSP System Toolbox) to Spectrogram or Spectrum and spectrogram.

Color-limits (minimum) — Spectrogram minimum

-80 (default) | scalar

Specify the signal power for the minimum color value of the spectrogram.

Tunable: Yes

Dependency

To use this parameter, set “View” (DSP System Toolbox) to Spectrogram or Spectrum and spectrogram.

Programmatic Use

See ColorLimits.

Color-limits (maximum) — Spectrogram maximum

20 (default) | scalar

Specify the signal power for the maximum color value of the spectrogram.

Tunable: Yes

Dependency

To use this parameter, set “View” (DSP System Toolbox) to Spectrogram or Spectrum and spectrogram.

Programmatic Use

See ColorLimits.

Style

The Style dialog box controls how to Spectrum Analyzer appears. To open the Style properties, in the Spectrum Analyzer menu, select View > Style or select the button in the toolbar dropdown.
**Figure color — Window background**  
gray (default) | color picker

Specify the color that you want to apply to the background of the scope figure.

**Plot type — Plot type**  
Line (default) | Stem

Specify whether to display a Line or Stem plot.

**Programmatic Use**

See `PlotType`.

**Axes colors — Axes background color**  
black (default) | color picker

Specify the color that you want to apply to the background of the axes.

**Properties for line — Channel for visual property settings**  
channel names

Specify the channel for which you want to modify the visibility, line properties, and marker properties.

**Visible — Channel visibility**  
on (default) | off

Specify whether the selected channel is visible. If you clear this check box, the line disappears. You can also change signal visibility using the scope legend.

**Line — Line style**  
line, 0.5, yellow (default)

Specify the line style, line width, and line color for the selected channel.

**Marker — Data point markers**  
none (default)

Specify marks for the selected channel to show at its data points. This parameter is similar to the 'Marker' property for plots. You can choose any of the marker symbols from the dropdown.
Axes Scaling

The Axes Scaling dialog box controls the axes limits of the Spectrum Analyzer. To open the Axes Scaling properties, in the Spectrum Analyzer menu, select Tools > Axes Scaling > Axes Scaling Properties.

Axes scaling/Color scaling — Automatic axes scaling
Auto (default) | Manual | After N Updates

Specify when the scope automatically scales the y-axis. If the spectrogram is displayed, specify when the scope automatically scales the color axis. By default, this parameter is set to Auto, and the scope does not shrink the y-axis limits when scaling the axes or color. You can select one of the following options:

• Auto — The scope scales the axes or color as needed, both during and after simulation. Selecting this option shows the Do not allow Y-axis limits to shrink or Do not allow color limits to shrink.
• Manual — When you select this option, the scope does not automatically scale the axes or color. You can manually scale the axes or color in any of the following ways:
  • Select Tools > Scaling Properties.
  • Press one of the Scale Axis Limits toolbar buttons.
  • When the scope figure is the active window, press Ctrl+A.
• After N Updates — Selecting this option causes the scope to scale the axes or color after a specified number of updates. This option is useful, and most efficient, when your frequency signal values quickly reach steady-state after a short period. Selecting this option shows the Number of updates edit box where you can modify the number of updates to wait before scaling.

Tunable: Yes

Programmatic Use

See AxesScaling.

Do not allow Y-axis/color limits to shrink — Axes scaling limits
on (default) | off

When you select this parameter, the y-axis is allowed to grow during axes scaling operations. If the spectrogram is displayed, selecting this parameter allows the color
limits to grow during axis scaling. If you clear this check box, the y-axis or color limits can shrink during axes scaling operations.

**Dependency**

This parameter appears only when you select Auto for the **Axis scaling** or **Color scaling** parameter. When you set the **Axes scaling** or **Color scaling** parameter to Manual or After N Updates, the y-axis or color limits can shrink.

**Number of updates — Number of updates before scaling**

10 (default) | positive number

The number of updates after which the axes scale, specified as a positive integer. If the spectrogram is displayed, this parameter specifies the number of updates after which the color axes scales.

**Tunable:** Yes

**Dependency**

This parameter appears only when you set “Axes scaling/Color scaling” (DSP System Toolbox) to After N Updates.

**Programmatic Use**

See AxesScalingNumUpdates.

**Scale limits at stop — Scale axes at stop**

off (default) | on

Select this check box to scale the axes when the simulation stops. If the spectrogram is displayed, select this check box to scale the color when the simulation stops. The y-axis is always scaled. The x-axis limits are only scaled if you also select the **Scale X-axis limits** check box.

**Data range (%) — Percent of axes**

100 (default) | number in the range [1,100]

Set the percentage of the axis that the scope uses to display the data when scaling the axes. If the spectrogram is displayed, set the percentage of the power values range within the colormap. Valid values are from 1 through 100. For example, if you set this parameter to 100, the scope scales the axis limits such that your data uses the entire axis range. If you then set this parameter to 30, the scope increases the y-axis or color range such that your data uses only 30% of the axis range.
**Tunable:** Yes

**Align — Alignment along axes**  
Center (default) | Bottom | Top | Left | Right

Specify where the scope aligns your data along the axis when it scales the axes. If the spectrogram is displayed, specify where the scope aligns your data along the axis when it scales the color. If you are using CCDF Measurements (DSP System Toolbox), the x axis is also configurable.

**Tunable:** Yes

### Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>double</th>
<th>enumerated</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Algorithms

**Welch's Method**

When you set the Method property to Welch, the following algorithms apply. The Spectrum Analyzer uses the RBW or the Window Length setting in the Spectrum Settings pane to determine the data window length. Then, it partitions the input signal into a number of windowed data segments. Finally, Spectrum Analyzer uses the modified periodogram method to compute spectral updates, averaging the windowed periodograms for each segment.
Spectrum Analyzer requires that a minimum number of samples to compute a spectral estimate. This number of input samples required to compute one spectral update is shown as **Samples/update** in the **Main options** pane. This value is directly related to resolution bandwidth, **RBW**, by the following equation, or to the window length, by the equation shown in step 2.

$$N_{samples} = \left(1 - \frac{O_p}{100}\right) \times \frac{NENBW \times F_s}{RBW}$$

The normalized effective noise bandwidth, **NENBW**, is a factor that depends on the windowing method. Spectrum Analyzer shows the value of **NENBW** in the **Window Options** pane of the **Spectrum Settings** pane. **Overlap percentage**, **$O_p$**, is the value of the **Overlap %** parameter in the **Window Options** pane of the **Spectrum Settings** pane. **$F_s$** is the sample rate of the input signal. Spectrum Analyzer shows sample rate in the **Main Options** pane of the **Spectrum Settings** pane.

1. When in **RBW (Hz)** mode, the window length required to compute one spectral update, **$N_{window}$**, is directly related to the resolution bandwidth and normalized effective noise bandwidth:

$$N_{window} = \frac{NENBW \times F_s}{RBW}$$

When in **Window Length** mode, the window length is used as specified.

2. The number of input samples required to compute one spectral update, **$N_{samples}$**, is directly related to the window length and the amount of overlap by the following equation.

$$N_{samples} = \left(1 - \frac{O_p}{100}\right)N_{window}$$

When you increase the overlap percentage, fewer new input samples are needed to compute a new spectral update. For example, if the window length is 100, then the number of input samples required to compute one spectral update is given as shown in the following table.

<table>
<thead>
<tr>
<th>$O_p$</th>
<th>$N_{samples}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0%</td>
<td>100</td>
</tr>
</tbody>
</table>
The normalized effective noise bandwidth, \( NENBW \), is a window parameter determined by the window length, \( N_{\text{window}} \), and the type of window used. If \( w(n) \) denotes the vector of \( N_{\text{window}} \) window coefficients, then \( NENBW \) is given by the following equation.

\[
NENBW = N_{\text{window}} \times \frac{\sum_{n=1}^{N_{\text{window}}} w^2(n)}{\left( \sum_{n=1}^{N_{\text{window}}} w(n) \right)^2}
\]

When in RBW (Hz) mode, you can set the resolution bandwidth using the value of the RBW (Hz) parameter on the Main options pane of the Spectrum Settings pane. You must specify a value to ensure that there are at least two RBW intervals over the specified frequency span. The ratio of the overall span to RBW must be greater than two:

\[
\frac{\text{span}}{\text{RBW}} > 2
\]

By default, the RBW (Hz) parameter on the Main options pane is set to Auto. In this case, the Spectrum Analyzer determines the appropriate value to ensure that there are 1024 RBW intervals over the specified frequency span. When you set RBW (Hz) to Auto, RBW is calculated as:

\[
\text{RBW}_{\text{auto}} = \frac{\text{span}}{1024}
\]

When in Window Length mode, you specify \( N_{\text{window}} \) and the resulting RBW is:

\[
\frac{NENBW \times F_s}{N_{\text{window}}}
\]

Sometimes, the number of input samples provided are not sufficient to achieve the resolution bandwidth that you specify. When this situation occurs, Spectrum Analyzer displays a message.
Spectrum Analyzer removes this message and displays a spectral estimate when enough data has been input.

**Note** The number of FFT points \( (N_{\text{fft}}) \) is independent of the window length \( (N_{\text{window}}) \). You can set them to different values if \( N_{\text{fft}} \) is greater than or equal to \( N_{\text{window}} \).

**Filter Bank**

When you set the **Method** property to **Filter Bank**, the following algorithms apply. The Spectrum Analyzer uses the **RBW (Hz)** or the **Number of frequency band** property in the **Spectrum Settings** pane to determine the input frame length.

Spectrum Analyzer requires a minimum number of samples to compute a spectral estimate. This number of input samples required to compute one spectral update is shown as **Samples/update** in the **Main options** pane. This value is directly related to resolution bandwidth, \( RBW \), by the following equation.

\[
N_{\text{samples}} = \frac{F_s}{RBW}
\]

\( F_s \) is the sample rate of the input signal. Spectrum Analyzer shows sample rate in the **Main Options** pane of the **Spectrum Settings** pane.

1. When in **RBW (Hz)** mode, you can set the resolution bandwidth using the value of the **RBW (Hz)** parameter on the **Main options** pane of the **Spectrum Settings** pane. You must specify a value to ensure that there are at least two RBW intervals over the specified frequency span. The ratio of the overall span to RBW must be greater than two:
\[ \frac{\text{span}}{\text{RBW}} > 2 \]

By default, the \textbf{RBW} parameter on the \textbf{Main options} pane is set to \textbf{Auto}. In this case, the Spectrum Analyzer determines the appropriate value to ensure that there are 1024 RBW intervals over the specified frequency span. Thus, when you set \textbf{RBW} to \textbf{Auto}, it is calculated by the following equation.

\[ \text{RBW}_{\text{auto}} = \frac{\text{span}}{1024} \]

2. When in \textbf{Number of frequency bands} mode, you specify the input frame size. When the number of frequency bands is \textbf{Auto}, the resulting RBW is:

\[ \text{RBW} = \frac{F_s}{\text{Input Frame Size}} \]

When the number of frequency bands is manually specified, the resulting RBW is:

\[ \text{RBW} = \frac{F_s}{\text{FFTLength}} \]

For more information about the filter bank algorithm, see “Polyphase Implementation” (DSP System Toolbox).

Sometimes, the number of input samples provided are not sufficient to achieve the resolution bandwidth that you specify. When this situation occurs, Spectrum Analyzer displays a message:

![The Spectrum Analyzer requires 75 samples to update the display. It will update once the required number of samples have been input.]

Spectrum Analyzer removes this message and displays a spectral estimate when enough data has been input.
Nyquist frequency interval

When the PlotAsTwoSidedSpectrum property is set to true, the interval is \( \left[ -\frac{\text{SampleRate}}{2}, \frac{\text{SampleRate}}{2} \right] + \text{FrequencyOffset} \) hertz.

When the PlotAsTwoSidedSpectrum property is set to false, the interval is \( \left[ 0, \frac{\text{SampleRate}}{2} \right] + \text{FrequencyOffset} \) hertz.

Periodogram and Spectrogram

Spectrum Analyzer calculates and plots the power spectrum, power spectrum density, and RMS computed by the modified Periodogram estimator. For more information about the Periodogram method, see periodogram.

Power Spectral Density — The power spectral density (PSD) is given by the following equation.

\[
\text{PSD}(f) = \frac{1}{P} \sum_{p=1}^{P} \left( \sum_{n=1}^{N_{\text{FFT}}} x_p[n] e^{-j2\pi f(n-1)T} \right)^2 \\
F_s \times \sum_{n=1}^{N_{\text{window}}} w[n]
\]

In this equation, \( x[n] \) is the discrete input signal. On every input signal frame, Spectrum Analyzer generates as many overlapping windows as possible, with each window denoted as \( x^{(p)}[n] \), and computes their periodograms. Spectrum Analyzer displays a running average of the \( P \) most current periodograms.

Power Spectrum — The power spectrum is the product of the power spectral density and the resolution bandwidth, as given by the following equation.
\[ P_{\text{spectrum}}(f) = \text{PSD}(f) \times RBW = \text{PSD}(f) \times \frac{F_s \times NENBW}{N_{\text{window}}} \]

\[
= \frac{1}{P} \sum_{p=1}^{P} \left| \sum_{n=1}^{N_{\text{FFT}}} x^p[n] e^{-j2\pi f(n-1)T} \right|^2 \left( \sum_{n=1}^{N_{\text{window}}} w[n] \right)^2
\]

**Frequency Vector**

When set to \textit{Auto}, the frequency vector for frequency-domain input is calculated by the software.

When the \textit{PlotAsTwoSidedSpectrum} property is set to true, the frequency vector is:

\[
\left[ -\frac{\text{SampleRate}}{2}, \frac{\text{SampleRate}}{2} \right]
\]

When the \textit{PlotAsTwoSidedSpectrum} property is set to false, the frequency vector is:

\[
\left[ 0, \frac{\text{SampleRate}}{2} \right]
\]

**Occupied BW**

The \textit{Occupied BW} is calculated as follows.

1. Calculate the total power in the measured frequency range.
2. Determine the lower frequency value. Starting at the lowest frequency in the range and moving upward, the power distributed in each frequency is summed until this result is

\[
100 - \frac{\text{OccupiedBW}}{2}
\]

of the total power.
3 Determine the upper frequency value. Starting at the highest frequency in the range and moving downward, the power distributed in each frequency is summed until the result reaches

\[
100 - \frac{\text{OccupiedBW\%}}{2}
\]

of the total power.

4 The bandwidth between the lower and upper power frequency values is the occupied bandwidth.

5 The frequency halfway between the lower and upper frequency values is the center frequency.

**Distortion Measurements**

The *Distortion Measurements* are computed as follows.

1 Spectral content is estimated by finding peaks in the spectrum. When the algorithm detects a peak, it records the width of the peak and clears all monotonically decreasing values. That is, the algorithm treats all these values as if they belong to the peak. Using this method, all spectral content centered at DC (0 Hz) is removed from the spectrum and the amount of bandwidth cleared \((W_0)\) is recorded.

2 The fundamental power \((P_1)\) is determined from the remaining maximum value of the displayed spectrum. A local estimate \((F_e_1)\) of the fundamental frequency is made by computing the central moment of the power near the peak. The bandwidth of the fundamental power content \((W_1)\) is recorded. Then, the power from the fundamental is removed as in step 1.

3 The power and width of the higher-order harmonics \((P_2, W_2, P_3, W_3, \text{etc.})\) are determined in succession by examining the frequencies closest to the appropriate multiple of the local estimate \((F_e_1)\). Any spectral content that decreases monotonically about the harmonic frequency is removed from the spectrum first before proceeding to the next harmonic.

4 Once the DC, fundamental, and harmonic content is removed from the spectrum, the power of the remaining spectrum is examined for its sum \((P_{\text{remaining}})\), peak value \((P_{\text{maxspur}})\), and median value \((P_{\text{esnoise}})\).

5 The sum of all the removed bandwidth is computed as \(W_{\text{sum}} = W_0 + W_1 + W_2 + \ldots + W_n\).
The sum of powers of the second and higher-order harmonics are computed as:
\[ P_{\text{harmonic}} = P_2 + P_3 + P_4 + \ldots + P_n. \]

6 The sum of the noise power is estimated as:
\[ P_{\text{noise}} = (P_{\text{remaining}} \cdot dF + P_{\text{est. noise}} \cdot W_{\text{sum}})/RBW \]
Where \( dF \) is the absolute difference between frequency bins, and \( RBW \) is the resolution bandwidth of the window.

7 The metrics for SNR, THD, SINAD, and SFDR are then computed from the estimates.

\[
THD = 10 \cdot \log_{10}\left(\frac{P_{\text{harmonic}}}{P_1}\right)
\]
\[
SINAD = 10 \cdot \log_{10}\left(\frac{P_1}{P_{\text{harmonic}} + P_{\text{noise}}}\right)
\]
\[
SNR = 10 \cdot \log_{10}\left(\frac{P_1}{P_{\text{noise}}}\right)
\]
\[
SFDR = 10 \cdot \log_{10}\left(\frac{P_1}{\max(P_{\text{maxspur}}, \max(P_2, P_3, \ldots, P_n))}\right)
\]

Harmonic Measurements

1 The harmonic distortion measurements use the spectrum trace shown in the display as the input to the measurements. The default Hann window setting of the Spectrum Analyzer may exhibit leakage that can completely mask the noise floor of the measured signal.
The harmonic measurements attempt to correct for leakage by ignoring all frequency content that decreases monotonically away from the maximum of harmonic peaks. If the window leakage covers more than 70% of the frequency bandwidth in your spectrum, you may see a blank reading (–) reported for **SNR** and **SINAD**. If your application can tolerate the increased equivalent noise bandwidth (ENBW), consider using a Kaiser window with a high attenuation (up to 330 dB) to minimize spectral leakage.
The DC component is ignored.

After windowing, the width of each harmonic component masks the noise power in the neighborhood of the fundamental frequency and harmonics. To estimate the noise power in each region, Spectrum Analyzer computes the median noise level in the nonharmonic areas of the spectrum. It then extrapolates that value into each region.

$N^{th}$ order intermodulation products occur at $A*F1 + B*F2$, where $F1$ and $F2$ are the sinusoid input frequencies and $|A| + |B| = N$. $A$ and $B$ are integer values.

For intermodulation measurements, the third-order intercept (TOI) point is computed as follows, where $P$ is power in decibels of the measured power referenced to 1 milliwatt (dBm):
\[ TOI_{\text{lower}} = P_{F1} + (P_{F2} - P_{(2F1,F2)})/2 \]
\[ TOI_{\text{upper}} = P_{F2} + (P_{F1} - P_{(2F2,F1)})/2 \]
\[ TOI = + (TOI_{\text{lower}} + TOI_{\text{upper}})/2 \]

**Averaging Method**

The moving averaging is done by using one of two methods:

- **Running** — For each frame of input, average the last \( N \) scaled \( Z \) vectors that are computed by the algorithm. The variable \( N \) is the value you specify for the number of spectral averages. If the algorithm does not have enough \( Z \) vectors, the algorithm uses zeros to fill the empty elements.

- **Exponential** — The moving average using the exponential weighting method is computed over the current \( Z \) vector and the previous \( Z \) vector. For details on the computation, see “Exponential Weighting Method” (DSP System Toolbox) in the \texttt{dsp.MovingAverage} object.

**Extended Capabilities**

**C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

This block can be used for simulation visibility in systems that generate code, but is not included in the generated code.

**HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

This block can be used for simulation visibility in subsystems that generate HDL code, but is not included in the hardware implementation.

**PLC Code Generation**

Generate Structured Text code using Simulink® PLC Coder™.
This block can be used for simulation visibility in systems that generate PLC code, but is not included in the generated code.

**Fixed-Point Conversion**
Design and simulate fixed-point systems using Fixed-Point Designer™.

This block accepts fixed-point input, but converts it to double for display.

**See Also**

**Objects**
SpectrumAnalyzerConfiguration | dsp.SpectrumAnalyzer

**Functions**
getMeasurementsData | getSpectralMaskStatus | getSpectrumData

**Blocks**
Array Plot | Time Scope

**Topics**
“Display Frequency-Domain Data in Spectrum Analyzer” (DSP System Toolbox)
“Spectral Analysis” (DSP System Toolbox)
“Display Frequency-Domain Data in Spectrum Analyzer” (DSP System Toolbox)

**Introduced in R2014b**
Slider Switch

Toggle parameter between two values
Library: Simulink / Dashboard

Description

The Slider Switch block toggles the value of the connected block parameter between two values during simulation. For example, you can connect the Slider Switch block to a Switch block in your model and change its state during simulation. Use the Slider Switch block with other Dashboard blocks to create an interactive dashboard for your model.

Double-clicking the Slider Switch block does not open its dialog box during simulation and when the block is selected. To edit the block’s parameters, you can use the Property Inspector, or you can right-click the block and select Block Parameters from the context menu.

Connecting Dashboard Blocks

Dashboard blocks do not use ports to make connections. To connect Dashboard blocks to variables and block parameters in your model, use connect mode. Connect mode facilitates the process of connecting Dashboard blocks in your model, especially when you want to connect multiple blocks at once. If you only want to connect a single Dashboard block, you can also use the Connection table in the block dialog box to make the connection.

Tip To facilitate understanding and debugging your model, you can modify Dashboard block connections in your model during normal and accelerator mode simulations.

Note Dashboard blocks cannot connect to variables until you update your model diagram. To connect Dashboard blocks to variables or modify variable values between opening your model and running a simulation, update your model diagram using Ctrl+D.
To enter connect mode, click the **Connect** button that appears above your unconnected Dashboard block when you pause on it.

In connect mode, when you select one or more signals or blocks, a list of parameters and signals available for connection appears. Select a signal or parameter from the list to connect the selected Dashboard block. To connect another Dashboard block, pause on the block and click the **Connect** button above it. Then, make a selection of signals and blocks in your model, and choose a signal or parameter to connect.
When you finish connecting the Dashboard blocks in your model, click the **Exit** button in the upper-right of the canvas to exit connect mode.

**Tip** You can hide the message shown on unconnected blocks using the `set_param` function with the `ShowInitialText` block parameter.

**Parameter Logging**

Tunable parameters connected to Dashboard blocks are logged to the Simulation Data Inspector, where you can view parameter values along with logged signal data. You can access logged parameter data in the MATLAB workspace by exporting the parameter data from the Simulation Data Inspector UI or by using the `Simulink.sdi.exportRun` function. For more information about exporting data with the Simulation Data Inspector UI, see “Export Data from the Simulation Data Inspector”. The parameter data is stored in a `Simulink.SimulationData.Parameter` object, accessible as an element in the exported `Simulink.SimulationData.Dataset`. 
Limitations

- Dashboard blocks can only connect to real scalar signals.
- You cannot use the Connection table to connect a Dashboard block to a block that is commented out. When you connect a Dashboard block to a commented block using connect mode, the Dashboard block does not display the connected value until you uncomment the block.
- Dashboard blocks cannot connect to signals inside referenced models.
- When you simulate a model hierarchy, Dashboard blocks inside referenced models do not update.
- Dashboard blocks do not support rapid accelerator simulation.
- Some signals are not available during accelerator mode simulations. To view such signals on a Dashboard block, mark the signal for logging.
- Parameters specified by indexing a variable array do not appear in the Connection table. For example, a block parameter defined as engine(1) using the variable engine does not appear in the table.

To access the parameter in the Connection table, assign the indexed value to a scalar variable, such as engine_1. Then, use the scalar variable to define the block parameter.
- When you connect a Dashboard block to a variable or parameter during simulation, the data for that variable or parameter does not get logged to the Simulation Data Inspector. To log variable and parameter data to the Simulation Data Inspector, connect the Dashboard block to the variable or parameter prior to simulation.
- When you simulate a model in external mode with the Default parameter behavior set to Inlined, Dashboard blocks can appear to change parameter and variable values. However, the change does not propagate to the simulation. For example, the Gain block displays changes made to its Gain parameter using the Dashboard blocks, but the Gain value used in the simulation does not change.

Parameters

Connection — Select a variable or block parameter to connect
variable and parameter connection options

Select the variable or block parameter to control using the Connection table. Populate the Connection table by selecting one or more blocks in your model. Select the radio
button next to the variable or parameter you want to control, and click **Apply**. To facilitate understanding and debugging your model, you can connect Dashboard blocks to variables and parameters in your model during simulation.

**Note** To see workspace variables in the connection table, update the model diagram using **Ctrl+D**.

### Programmatic Use

To programmatically connect a Dashboard block to a tunable parameter or a variable, use a `Simulink.HMI.ParamSourceInfo` object. The `Simulink.HMI.ParamSourceInfo` object contains four properties. Some of the properties apply to connecting Dashboard blocks to parameters, and some apply to connecting Dashboard blocks to variables. Not all fields have a value for a connection because a given Dashboard block connects to either a parameter or a variable.

**Block Parameter:** **Binding**  
**Type:** `Simulink.HMI.ParamSourceInfo`  
**Default:** `[]`

**States — Pair values and labels**  
Scalar and character vector

Pairs of values to assign to the connected variable or parameter and text to display on the block. Switches have two states — **Left** and **Right** — one corresponding to each switch position. Each state contains a **Value** and a **Label**.

- **Value** — Value to assign to the connected variable or parameter when the switch is in the corresponding position.
- **Label** — Text to display on the block for the corresponding position.

This table describes the default configuration for the block.

**States**

<table>
<thead>
<tr>
<th>Position</th>
<th>State Value</th>
<th>State Label</th>
</tr>
</thead>
<tbody>
<tr>
<td>Left</td>
<td>0</td>
<td>on</td>
</tr>
<tr>
<td>Right</td>
<td>1</td>
<td>off</td>
</tr>
</tbody>
</table>
Programmatic Use

To configure the **States** for the block programmatically, specify the value of the **States** parameter as a structure array containing two elements with fields:

- **Value** — Scalar double value for the state.
- **Label** — String or character array to use as the label for the switch position.

```matlab
leftState.Value = 0;
leftState.Label = 'Off';
rightState.Value = 1;
rightState.Label = 'On';
switchStates = [leftState rightState];
```

**Block Parameter:** States  
**Type:** two element array of structures

**Label — Block label position**  
**Top** (default) | **Bottom** | **Hide**

Position of the block label. When the block is connected to an element in the model, the label is the name of the connected element.

Programmatic Use

**Block Parameter:** LabelPosition  
**Type:** character vector  
**Values:** 'Top' | 'Bottom' | 'Hide'  
**Default:** 'Top'

Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>double</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>no</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Extended Capabilities

#### C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

Dashboard blocks are ignored for code generation.

#### See Also
Rocker Switch | Rotary Switch | Toggle Switch

#### Topics
“Tune and Visualize Your Model with Dashboard Blocks”
“Decide How to Visualize Simulation Data”

**Introduced in R2015a**

<table>
<thead>
<tr>
<th>Zero-Crossing Detection</th>
<th>no</th>
</tr>
</thead>
</table>
Sqrt

Calculate square root, signed square root, or reciprocal of square root

Library:  Simulink / Math Operations
         HDL Coder / HDL Floating Point Operations
         HDL Coder / Math Operations

Description

The Sqrt block calculates the square root, signed square root, or reciprocal of square root on the input signal. Select one of the following functions from the Function parameter list.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Mathematical Expression</th>
<th>MATLAB Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>sqrt</td>
<td>Square root of the input</td>
<td>$u^{0.5}$</td>
<td>sqrt</td>
</tr>
<tr>
<td>signedSqrt</td>
<td>Square root of the absolute value of the input, multiplied by the sign of the input</td>
<td>$\text{sign}(u)\times</td>
<td>u</td>
</tr>
<tr>
<td>rSqrt</td>
<td>Reciprocal of the square root of the input</td>
<td>$u^{-0.5}$</td>
<td>—</td>
</tr>
</tbody>
</table>

The block icon changes to match the function.

Ports

Input

Port_1 — Input signal

scalar | vector | matrix
Input signal to the block to calculate the square root, signed square root, or reciprocal of square root. The sqrt function accepts real or complex inputs, except for complex fixed-point signals. signedSqrt and rSqrt do not accept complex inputs. For the signedSqrt function, the input signal must be a floating point number.

If the input is negative, set the **Output signal** to complex for all functions except signedSqrt.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

**Output**

**Port_1 — Output signal**

```
scalar | vector | matrix
```

Output signal that is the square root, signed square root, or reciprocal of square root of the input signal. When the input is an integer or fixed-point type, the output must be floating point.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

**Parameters**

**Main**

**Function — Function the block performs**

```
sqrt (default) | signedSqrt | rSqrt
```

Specify the mathematical function that the block calculates. The block icon changes to match the function you select.

<table>
<thead>
<tr>
<th>Function</th>
<th>Block Icon</th>
</tr>
</thead>
<tbody>
<tr>
<td>sqrt</td>
<td><img src="image" alt="sqrt Icon" /></td>
</tr>
<tr>
<td>Function</td>
<td>Block Icon</td>
</tr>
<tr>
<td>--------------</td>
<td>------------</td>
</tr>
<tr>
<td>signedSqrt</td>
<td>![signedSqrt Icon]</td>
</tr>
<tr>
<td>rSqrt</td>
<td>![rSqrt Icon]</td>
</tr>
</tbody>
</table>

**Dependency**

When this parameter is set to `signedSqrt`, the Intermediate results data type parameter is disabled.

**Programmatic Use**

Block Parameter: Operator  
Type: character vector  
Values: 'sqrt' | 'signedSqrt' | 'rSqrt'  
Default: 'sqrt'

**Output signal type — Output signal type**

auto (default) | real | complex

Specify the output signal type of the block.

<table>
<thead>
<tr>
<th>Function</th>
<th>Input Signal Type</th>
<th>Output Signal Type</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Auto</td>
<td>Real</td>
</tr>
<tr>
<td>sqrt</td>
<td>real</td>
<td>real for nonnegative inputs</td>
</tr>
<tr>
<td></td>
<td>real</td>
<td>NaN for negative inputs</td>
</tr>
<tr>
<td></td>
<td>complex</td>
<td>complex</td>
</tr>
<tr>
<td>signedSqrt</td>
<td>real</td>
<td>real</td>
</tr>
<tr>
<td></td>
<td>complex</td>
<td>error</td>
</tr>
<tr>
<td>rSqrt</td>
<td>real</td>
<td>real</td>
</tr>
<tr>
<td></td>
<td>complex</td>
<td>error</td>
</tr>
</tbody>
</table>
**Programmatic Use**

**Block Parameter:** OutputSignalType  
**Type:** character vector  
**Values:** 'auto' | 'real' | 'complex'  
**Default:** 'auto'

**Sample time — Specify sample time as a value other than -1**

-1 (default) | scalar | vector

Specify the sample time as a value other than -1. For more information, see “Specify Sample Time”.

**Dependencies**

This parameter is not visible unless it is explicitly set to a value other than -1. To learn more, see “Blocks for Which Sample Time Is Not Recommended”.

**Programmatic Use**

**Block Parameter:** SampleTime  
**Type:** character vector  
**Values:** scalar or vector  
**Default:** ' -1'

**Algorithm**

**Method — Method to compute reciprocal of square root**

Exact (default) | Newton-Raphson

Specify the method for computing the reciprocal of a square root. This parameter is only valid for the rSqrt function.

<table>
<thead>
<tr>
<th>Method</th>
<th>Data Types Supported</th>
<th>When to Use This Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exact</td>
<td>Floating point</td>
<td>You do not want an approximation.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Note</strong> The input or output must be floating point.</td>
</tr>
<tr>
<td>Newton-Raphson</td>
<td>Floating-point, fixed-point, and built-in integer types</td>
<td>You want a fast, approximate calculation.</td>
</tr>
</tbody>
</table>

The Exact method provides results that are consistent with MATLAB computations.
The algorithms for sqrt and signedSqrt are always of Exact type, no matter what selection appears on the block dialog box.

Programmatic Use
Block Parameter: AlgorithmType
Type: character vector
Values: 'Exact' | 'Newton-Raphson'
Default: 'Exact'

Number of iterations — Number of iterations used for Newton Raphson algorithm
3 (default) | integer

Specify the number of iterations to perform the Newton-Raphson algorithm. This parameter is valid with the rSqrt function and the Newton-Raphson value for Method.

Note If you enter 0, the block output is the initial guess of the Newton-Raphson algorithm.

Programmatic Use
Block Parameter: Iterations
Type: character vector
Values: integer
Default: '3'

Data Types

Click the Show data type assistant button to display the Data Type Assistant, which helps you set the data type attributes. For more information, see “Specify Data Types Using Data Type Assistant”.

Intermediate results data type — Data type of intermediate results
Inherit:Inherit via internal rule (default) | Inherit: Inherit from input | Inherit: Inherit from output | double | single | int8 | uint8 | int16 | uint16 | int32 | uint32 | int64 | uint64 | fixdt(1,16,,0) | fixdt(1,16,2^0,0) | <data type expression>
Specify the data type for intermediate results when you set Function to \texttt{sqrt} or \texttt{rSqrt} on the \texttt{Main} pane.

The type can be inherited, specified directly, or expressed as a data type object such as \texttt{Simulink.NumericType}.

Follow these guidelines on setting an intermediate data type explicitly for the square root function, \texttt{sqrt}:

<table>
<thead>
<tr>
<th>Input and Output Data Types</th>
<th>Intermediate Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input or output is double.</td>
<td>Use double.</td>
</tr>
<tr>
<td>Input or output is single, and any non-single data type is \textit{not} double.</td>
<td>Use single or double.</td>
</tr>
<tr>
<td>Input and output are fixed point.</td>
<td>Use fixed point.</td>
</tr>
</tbody>
</table>

Follow these guidelines on setting an intermediate data type explicitly for the reciprocal square root function, \texttt{rSqrt}:

<table>
<thead>
<tr>
<th>Input and Output Data Types</th>
<th>Intermediate Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input is double and output is not single.</td>
<td>Use double.</td>
</tr>
<tr>
<td>Input is not single and output is double.</td>
<td>Use double.</td>
</tr>
<tr>
<td>Input and output are fixed point.</td>
<td>Use fixed point.</td>
</tr>
</tbody>
</table>

\textbf{Caution} Do not set \textit{Intermediate results data type} to \texttt{Inherit:Inherit from output} when:

- You select \texttt{Newton-Raphson} to compute the reciprocal of a square root.
- The input data type is floating point.
- The output data type is fixed point.

Under these conditions, selecting \texttt{Inherit:Inherit from output} yields suboptimal performance and produces an error.

To avoid this error, convert the input signal from a floating-point to fixed-point data type. For example, insert a Data Type Conversion block in front of the Sqrt block to perform the conversion.
Dependency

This parameter is disabled when the Function parameter is set to signedSqrt.

Programmatic Use

**Block Parameter:** IntermediateResultsDataTypeStr  
**Type:** character vector  
**Values:**  
- 'Inherit: Inherit via internal rule'  
- 'Inherit: Inherit from input'  
- 'Inherit: Inherit from output'  
- 'double'  
- 'single'  
- 'int8', 'uint8', 'int16', 'uint16', 'int32', 'uint32', 'int64', 'uint64', fixdt(1,16,0), fixdt(1,16,2^0,0), '<data type expression>'  
**Default:** 'Inherit: Inherit via internal rule'

Output — Output data type

**Inherit:**  
- Same as first input (default)  
- Inherit via internal rule  
- Inherit via back propagation  
- double  
- single  
- int8  
- int16  
- uint8  
- uint16  
- int32  
- uint32  
- int64  
- uint64  
- fixdt(1,16,0), fixdt(1,16,2^0,0), fixdt(1,16,2^0,0), '<data type expression>'  
**Default:** 'Inherit: Inherit via internal rule'

Specify the output data type. The type can be inherited, specified directly, or expressed as a data type object such as Simulink.NumericType.

**Programmatic Use**

**Block Parameter:** OutDataTypeStr  
**Type:** character vector  
**Values:**  
- 'Inherit: Inherit via internal rule'  
- 'Inherit: Inherit via back propagation'  
- 'Inherit: Same as first input'  
- 'double'  
- 'single'  
- 'int8', 'uint8', 'int16', 'uint16', 'int32', 'uint32', 'int64', 'uint64', fixdt(1,16,0), fixdt(1,16,2^0,0), fixdt(1,16,2^0,0), '<data type expression>'  
**Default:** 'Inherit: Same as first input'

Minimum — Minimum output value for range checking

**[]** (default) | **scalar**

Specify the lower value of the output range that Simulink checks as a finite, real, double, scalar value.

**Note**  
If you specify a bus object as the data type for this block, do not set the minimum value for bus data on the block. Simulink ignores this setting. Instead, set the minimum values for bus elements of the bus object specified as the data type. For information on the Minimum parameter for a bus element, see Simulink.BusElement.
Simulink uses the minimum to perform:

- Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters”) for some blocks.
- Simulation range checking (see “Signal Ranges” and “Enable Simulation Range Checking”).
- Automatic scaling of fixed-point data types.
- Optimization of the code that you generate from the model. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. For more information, see “Optimize using the specified minimum and maximum values” (Simulink Coder).

**Note** Output minimum does not saturate or clip the actual output signal. Use the Saturation block instead.

**Programmatic Use**

**Block Parameter:** `OutMin`  
**Type:** character vector  
**Values:** scalar  
**Default:** `'[ ]'`

**Maximum — Maximum output value for range checking**  
`[]` (default) | scalar

Specify the upper value of the output range that Simulink checks as a finite, real, double, scalar value.

**Note** If you specify a bus object as the data type for this block, do not set the maximum value for bus data on the block. Simulink ignores this setting. Instead, set the maximum values for bus elements of the bus object specified as the data type. For information on the Maximum parameter for a bus element, see `Simulink.BusElement`.

Simulink uses the maximum value to perform:

- Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters”) for some blocks.
• Simulation range checking (see “Signal Ranges” and “Enable Simulation Range Checking”).
• Automatic scaling of fixed-point data types.
• Optimization of the code that you generate from the model. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. For more information, see “Optimize using the specified minimum and maximum values” (Simulink Coder).

**Note** Output maximum does not saturate or clip the actual output signal. Use the Saturation block instead.

**Programmatic Use**
**Block Parameter:** OutMax
**Type:** character vector
**Values:** scalar
**Default:** ['[ ]']

**Integer rounding mode — Rounding mode for fixed-point operations**
Floor (default) | Ceiling | Convergent | Nearest | Round | Simplest | Zero

Specify the rounding mode for fixed-point operations. For more information, see “Rounding” (Fixed-Point Designer).

**Programmatic Use**
**Block Parameter:** RndMeth
**Type:** character vector
**Values:** 'Ceiling' | 'Convergent' | 'Floor' | 'Nearest' | 'Round' | 'Simplest' | 'Zero'
**Default:** 'Floor'

**Lock output data type setting against changes by the fixed-point tools — Prevent fixed-point tools from overriding data types**
off (default) | on

Select to lock the output data type setting of this block against changes by the Fixed-Point Tool and the Fixed-Point Advisor. For more information, see “Use Lock Output Data Type Setting” (Fixed-Point Designer).
**Programmatic Use**

**Block Parameter:** LockScale  
**Type:** character vector  
**Values:** 'off' | 'on'  
**Default:** 'off'

**Saturate on integer overflow — Choose the behavior when integer overflow occurs**  
on (default) | boolean

<table>
<thead>
<tr>
<th>Action</th>
<th>Reasons for Taking This Action</th>
<th>What Happens for Overflows</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Select this check box.</td>
<td>Your model has possible overflow, and you want explicit saturation protection in the generated code.</td>
<td>Overflows saturate to either the minimum or maximum value that the data type can represent.</td>
<td>The maximum value that the int8 (signed, 8-bit integer) data type can represent is 127. Any block operation result greater than this maximum value causes overflow of the 8-bit integer. With the check box selected, the block output saturates at 127. Similarly, the block output saturates at a minimum output value of -128.</td>
</tr>
<tr>
<td>Action</td>
<td>Reasons for Taking This Action</td>
<td>What Happens for Overflows</td>
<td>Example</td>
</tr>
<tr>
<td>--------</td>
<td>--------------------------------</td>
<td>-----------------------------</td>
<td>---------</td>
</tr>
<tr>
<td>Do not select this check box.</td>
<td>You want to optimize efficiency of your generated code. You want to avoid overspecifying how a block handles out-of-range signals. For more information, see “Check for Signal Range Errors”.</td>
<td>Overflows wrap to the appropriate value that is representable by the data type.</td>
<td>The maximum value that the int8 (signed, 8-bit integer) data type can represent is 127. Any block operation result greater than this maximum value causes overflow of the 8-bit integer. With the check box cleared, the software interprets the overflow-causing value as int8, which can produce an unintended result. For example, a block result of 130 (binary 1000 0010) expressed as int8, is -126.</td>
</tr>
</tbody>
</table>

When you select this check box, saturation applies to every internal operation on the block, not just the output or result. Usually, the code generation process can detect when overflow is not possible. In this case, the code generator does not produce saturation code.

**Programmatic Use**

**Block Parameter:** SaturateOnIntegerOverflow  
**Type:** character vector  
**Value:** 'off' | 'on'  
**Default:** 'on'

**Block Characteristics**

<table>
<thead>
<tr>
<th>Data Types</th>
<th>double</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Multidimensional Signals
Yes

### Variable-Size Signals
Yes

### Zero-Crossing Detection
No

## Extended Capabilities

### C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

### HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

### Sqrt and Rsqrt Function Modes

For the Sqrt block with Function set to sqrt or rsqrt, the code generator supports various architectures and data types. The sqrtfunction architecture supports code generation in native floating-point mode. For this architecture, you can specify the HandleDenormals and LatencyStrategy settings from the Native Floating Point tab in the HDL Block Properties dialog box.

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Fixed-Point</th>
<th>Native Floating-Point</th>
<th>HandleDenormal s</th>
<th>LatencyStrategy</th>
</tr>
</thead>
<tbody>
<tr>
<td>sqrtfunction</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>sqrtnewton</td>
<td>✓</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>sqrtnewtonsingle</td>
<td>✓</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>
### Architecture

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Fixed-Point</th>
<th>Native Floating-Point</th>
<th>HandleDenormal</th>
</tr>
</thead>
<tbody>
<tr>
<td>sqrtbitset</td>
<td>✓</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>recipsqrtnewton</td>
<td>✓</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>recipsqrtnewtonsinglerate</td>
<td>✓</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>

### HDL Architecture

This block has multi-cycle implementations that introduce additional latency in the generated code. To see the added latency, view the generated model or validation model. See “Generated Model and Validation Model” (HDL Coder).

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Parameter</th>
<th>Additional cycles of latency</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SqrtFunction</td>
<td>None</td>
<td>0</td>
<td>Use a bitset shift/addition algorithm. The SqrtFunction architecture is equivalent to the SqrtBitset architecture with UseMultiplier set to off.</td>
</tr>
</tbody>
</table>
| SqrtBitset       | UseMultiplier | 0                            | Algorithm depends on the UseMultiplier setting:  
• off (default): Use a bitset shift/addition algorithm.  
• on: Use a multiply/add algorithm. |
<p>| SqrtNewton       | Iterations | Iterations + 3              | Use the iterative Newton method. Select this option to optimize area. The default value for Iterations is 3. The recommended value for Iterations is from 2 through 10. If Iterations is outside the recommended range, HDL Coder generates a message. |</p>
<table>
<thead>
<tr>
<th>Architecture</th>
<th>Parameter</th>
<th>Additional cycles of latency</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SqrtNewtonSingleRate</td>
<td>Iterations</td>
<td>(Iterations * 4) + 6</td>
<td>Use the single rate pipelined Newton method. Select this option to optimize speed, or if you want a single rate implementation. The default value for Iterations is 3. The recommended value for Iterations is from 2 through 10. If Iterations is outside the recommended range, the coder generates a message.</td>
</tr>
<tr>
<td>RecipSqrtNewton</td>
<td>Iterations</td>
<td>Iterations + 2</td>
<td>Use the iterative Newton method. Select this option to optimize area.</td>
</tr>
<tr>
<td>RecipSqrtNewtonSingleRate</td>
<td>Iterations</td>
<td>(Iterations * 4) + 5</td>
<td>Use the single rate pipelined Newton method. Select this option to optimize speed, or if you want a single rate implementation.</td>
</tr>
</tbody>
</table>

### HDL Block Properties

#### General

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConstrainedOutputPipeline</td>
<td>Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. For more details, see “ConstrainedOutputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td>Iterations</td>
<td>Number of iterations for SqrtNewton or SqrtNewtonSingleRate implementation.</td>
</tr>
<tr>
<td>InputPipeline</td>
<td>Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “InputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td>OutputPipeline</td>
<td>Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “OutputPipeline” (HDL Coder).</td>
</tr>
</tbody>
</table>
General

| UseMultiplier | Select algorithm for SqrtBitset implementation. The default is off. |

Note The Sqrt block does not support HDL code generation with double data types in the Native Floating Point mode.

Native Floating Point

| HandleDenormals | Specify whether you want HDL Coder to insert additional logic to handle denormal numbers in your design. Denormal numbers are numbers that have magnitudes less than the smallest floating-point number that can be represented without leading zeros in the mantissa. The default is inherit. See also “HandleDenormals” (HDL Coder). |
| LatencyStrategy | Specify whether to map the blocks in your design to inherit, Max, Min, Zero, or Custom for the floating-point operator. The default is inherit. See also “LatencyStrategy” (HDL Coder). |
| NFPCustomLatency | To specify a value, set LatencyStrategy to Custom. HDL Coder adds latency equal to the value that you specify for the NFPCustomLatency setting. See also “NFPCustomLatency” (HDL Coder). |

Restrictions

- Input must be an unsigned scalar value.
- Output is a fixed-point scalar value.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

Design and simulate fixed-point systems using Fixed-Point Designer™.
See Also
Math Function | Trigonometric Function

Introduced in R2010a
Squeeze

Remove singleton dimensions from multidimensional signal

Library: Simulink / Math Operations

Description

The Squeeze block removes singleton dimensions from its multidimensional input signal. A singleton dimension is any dimension whose size is one. The Squeeze block operates only on signals whose number of dimensions is greater than two. Scalar, vector, and matrix signals pass through the Squeeze block unchanged.

Ports

Input

Port_1 — Multidimensional input signal
multidimensional signal

Input signal that has any singleton dimensions removed in the output.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus

Output

Port_1 — Output signal with no singleton dimensions
multidimensional signal

Output signal with no singleton dimensions. For example, a multidimensional array of size 3-by-1-by-2 changes into a 3-by-2 signal. If there are no singleton dimensions in the input, then the input signal is passed through unchanged to the output.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus
Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>double</th>
<th>enumerated</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

PLC Code Generation
Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

See Also
Reshape

Introduced in R2007b
State Reader

Read a block state
Library: Simulink / Signal Routing

Description

The State Reader block reads the current state of a supported state owner block. When creating a library block with a State Reader block also include the state owner block.

Add a State Reader block to your model from the Simulink Library Browser.

State Reader blocks can read state from these state owner blocks:

- Discrete State-Space
- Discrete-Time Integrator
- Delay
- Unit Delay
- Discrete Transfer Fcn
- Discrete Filter
- Discrete FIR Filter
- Integrator
- Second-Order Integrator
- Conditional subsystem blocks such as Enabled Subsystem, Triggered Subsystem, and Function-Call Subsystem.
- S-Function with multiple discrete states.

To configure an S-Function block as a state owner block, exactly one data type work vector must be declared as discrete state vector using ssSetDWorkUsedAsDState and named using ssSetDWorkName or ssSetDWorkRTWIdentifier in mdlInitializeSizes.
Ports

Output

**Out** — State value

scalar | vector

State value read from a state owner block.

The dimension of the output is the dimension of the full state vector. Refer to the **Initial conditions** parameter for specific blocks. For example, for a Delay block with a **Delay length** of \(N\), the State Reader block returns a state vector of length \([1 \times N]\).

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

Parameters

**State owner block** — Show the state owner block

none (default)

Show the state owner block whose state this block is reading. To change the state owner block, select a block from the **State Owner Selector Tree**.

**Programmatic Use**

**Block Parameter:** StateOwnerBlock

**Type:** character vector

**Value:** '' | '<model path/block name>'

**Default:** ''

Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>bus</th>
<th>double</th>
<th>enumerated</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

See Also
Event Listener | Initialize Function | Reset Function | State Writer | Terminate Function

Topics
“Using Initialize, Reset, and Terminate Functions”
“Create Test Harness to Generate Function Calls”
State Writer

Write to a block state

Library: Simulink / Signal Routing

Description

The State Writer block sets the state of a supported state owner block. When creating a library block with a State Writer block also include the state owner block.

Add a State Writer block to your model from the Simulink Library Browser.

State Writer blocks can write state to these state owner blocks:

- Discrete State-Space
- Discrete-Time Integrator
- Delay
- Unit Delay
- Discrete Transfer Fcn
- Discrete Filter
- Discrete FIR Filter
- Integrator
- Second-Order Integrator
- Conditional subsystem blocks such as Enabled Subsystem, Triggered Subsystem, and Function-Call Subsystem.
- S-Function with multiple discrete states.

To configure an S-Function block as a state owner block, exactly one data type work vector must be declared as discrete state vector using ssSetDWorkUsedAsDState and named using ssSetDWorkName or ssSetDWorkRTWIdentifier in mdlInitializeSizes.
Ports

Input

In — State value
scalar | vector

State value written to a state owner block.

When writing to a state owner block with an input scalar, the scalar value is expanded to match the dimension of the state. All elements of the state are set to the same value.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

Parameters

State owner block — Show the state owner block
none (default)

Show the state owner block whose state this block is writing. To change the state owner block, select a block from the State Owner Selector Tree.

Programmatic Use
Block Parameter: StateOwnerBlock
Type: character vector
Value: '' | '<model path/block name>'
Default: ''

Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>bus</th>
<th>double</th>
<th>enumerated</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1-1977
Multidimensional Signals | yes
|
Variable-Size Signals | no
|
Zero-Crossing Detection | no

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

See Also
Event Listener | Initialize Function | Reset Function | State Reader | Terminate Function

Topics
“Using Initialize, Reset, and Terminate Functions”
“Create Test Harness to Generate Function Calls”
State-Space

Implement linear state-space system
Library: Simulink / Continuous

Description

The State-Space block implements a system whose behavior you define as

\[
\begin{align*}
\dot{x} &= Ax + Bu \\
y &= Cx + Du \\
x|_{t=t_0} &= x_0,
\end{align*}
\]

where \( x \) is the state vector, \( u \) is the input vector, \( y \) is the output vector, and \( x_0 \) is the initial condition of the state vector. The matrix coefficients must have these characteristics:

- \( A \) must be an \( n \)-by-\( n \) matrix, where \( n \) is the number of states.
- \( B \) must be an \( n \)-by-\( m \) matrix, where \( m \) is the number of inputs.
- \( C \) must be an \( r \)-by-\( n \) matrix, where \( r \) is the number of outputs.
- \( D \) must be an \( r \)-by-\( m \) matrix.

In general, the block has one input port and one output port. The number of rows in \( C \) or \( D \) matrix is the same as the width of the output port. The number of columns in the \( B \) or \( D \) matrix are the same as the width of the input port. If you want to model an autonomous linear system with no inputs, set the \( B \) and \( D \) matrices to empty. In this case, the block acts as a source block with no input port and one output port, and implements the following system:

\[
\begin{align*}
\dot{x} &= Ax \\
y &= Cx \\
x|_{t=t_0} &= x_0.
\end{align*}
\]
Simulink software converts a matrix containing zeros to a sparse matrix for efficient multiplication.

**Ports**

**Input**

Port_1 — Input signal
scalar | vector

Real-valued input vector of type double, where the width equals the number of columns in the B and D matrices. For more information, see “Description” on page 1-1979.

Data Types: double

**Output**

Port_1 — Output vector
scalar | vector

Real-valued output vector of data type double, with width equal to the number of rows in the C and D matrices. For more information, see “Description” on page 1-1979.

Data Types: double

**Parameters**

A — Matrix coefficient, A
1 (default) | scalar | vector | matrix

Specify the matrix coefficient A, as a real-valued n-by-n matrix, where n is the number of states. For more information on the matrix coefficients, see “Description” on page 1-1979.

**Programmatic Use**

Block Parameter: A
Type: character vector, string
Values: scalar | vector | matrix
Default: '1'
**B — Matrix coefficient, B**

1 (default) | scalar | vector | matrix

Specify the matrix coefficient $B$, as a real-valued $n$-by-$m$ matrix, where $n$ is the number of states and $m$ is the number of inputs. For more information on the matrix coefficients, see “Description” on page 1-1979.

**Programmatic Use**

**Block Parameter:** $B$

Type: character vector, string

Values: scalar | vector | matrix

Default: '1'

**C — Matrix coefficient, C**

1 (default) | scalar | vector | matrix

Specify the matrix coefficient $C$ as a real-valued $r$-by-$n$ matrix, where $r$ is the number of outputs and $n$ is the number of states. For more information on the matrix coefficients, see “Description” on page 1-1979.

**Programmatic Use**

**Block Parameter:** $C$

Type: character vector, string

Values: scalar | vector | matrix

Default: '1'

**D — Matrix coefficient, D**

1 (default) | scalar | vector | matrix

Specify the matrix coefficient $D$ as a real-valued $r$-by-$m$ matrix, where $r$ is the number of outputs and $m$ is the number of inputs. For more information on the matrix coefficients, see “Description” on page 1-1979.

**Programmatic Use**

**Block Parameter:** $D$

Type: character vector, string

Values: scalar | vector | matrix

Default: '1'

**Initial conditions — Initial state vector**

0 (default) | scalar | vector

Specify the initial state vector.
Limitations

The initial conditions of this block cannot be \texttt{inf} or \texttt{NaN}.

Programmatic Use
Block Parameter: \texttt{X0}
Type: character vector, string
Values: scalar | vector
Default: '0'

Absolute tolerance — Absolute tolerance for computing block states
auto (default) | scalar | vector

Absolute tolerance for computing block states, specified as a positive, real-valued, scalar or vector. To inherit the absolute tolerance from the Configuration Parameters, specify auto or -1.

- If you enter a real scalar, then that value overrides the absolute tolerance in the Configuration Parameters dialog box for computing all block states.
- If you enter a real vector, then the dimension of that vector must match the dimension of the continuous states in the block. These values override the absolute tolerance in the Configuration Parameters dialog box.
- If you enter auto or -1, then Simulink uses the absolute tolerance value in the Configuration Parameters dialog box (see “Solver Pane”) to compute block states.

Programmatic Use
Block Parameter: \texttt{AbsoluteTolerance}
Type: character vector, string
Values: 'auto' | '-1' | any positive real-valued scalar or vector
Default: 'auto'

State Name (e.g., 'position') — Assign unique name to each state
' ' (default) | 'position' | {'a', 'b', 'c'} | a | ...

Assign a unique name to each state. If this field is blank (' '), no name assignment occurs.

- To assign a name to a single state, enter the name between quotes, for example, 'position'.
- To assign names to multiple states, enter a comma-delimited list surrounded by braces, for example, { 'a', 'b', 'c'}. Each name must be unique.
To assign state names with a variable in the MATLAB workspace, enter the variable without quotes. A variable can be a character vector, string, cell array, or structure.

**Limitations**

- The state names apply only to the selected block.
- The number of states must divide evenly among the number of state names.
- You can specify fewer names than states, but you cannot specify more names than states.

For example, you can specify two names in a system with four states. The first name applies to the first two states and the second name to the last two states.

**Programmatic Use**

**Block Parameter:** ContinuousStateAttributes  
**Type:** character vector, string  
**Values:** ' ' | user-defined  
**Default:** ' '

**Block Characteristics**

<table>
<thead>
<tr>
<th>Data Types</th>
<th>double</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>yes</td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>no</td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
</tr>
</tbody>
</table>
Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

Not recommended for production-quality code. Relates to resource limits and restrictions on speed and memory often found in embedded systems. The code generated can contain dynamic allocation and freeing of memory, recursion, additional memory overhead, and widely-varying execution times. While the code is functionally valid and generally acceptable in resource-rich environments, smaller embedded targets often cannot support such code.

In general, consider using the Simulink Model Discretizer to map continuous blocks into discrete equivalents that support production code generation. To start the Model Discretizer, select Analysis > Control Design > Model Discretizer. One exception is the Second-Order Integrator block because, for this block, the Model Discretizer produces an approximate discretization.

See Also
Discrete State-Space | Transfer Fcn

Topics
“State”

Introduced before R2006a
Step

Generate step function

Library: Simulink / Sources

Description

The Step block provides a step between two definable levels at a specified time. If the simulation time is less than the Step time parameter value, the block's output is the Initial value parameter value. For simulation time greater than or equal to the Step time, the output is the Final value parameter value.

The numeric block parameters must be of the same dimensions after scalar expansion. If the Interpret vector parameters as 1-D option is off, the block outputs a signal of the same dimensions and dimensionality as the parameters. If the Interpret vector parameters as 1-D option is on and the numeric parameters are row or column vectors (that is, single-row or column 2-D arrays), the block outputs a vector (1-D array) signal. Otherwise, the block outputs a signal of the same dimensionality and dimensions as the parameters.

Ports

Output

Port_1 — Output step signal
scalar | vector

Output step function signal defined by the parameters Step time, Initial value, and Final value.

Data Types: double
Parameters

Step time — Time when step occurs
1 (default) | scalar

Specify the time, in seconds, when the output jumps from the Initial value parameter to the Final value parameter.

Programmatic Use
Block Parameter: Time
Type: character vector
Values: '1' | real- or complex-valued scalar, vector, or matrix
Default: '1'

Initial value — Output value before step
0 (default) | scalar

Specify the block output until the simulation time reaches the Step time parameter.

Programmatic Use
Block Parameter: Before
Type: character vector
Values: '1' | real- or complex-valued scalar, vector, or matrix
Default: '0'

Final value — Output value after step
1 (default) | scalar

Specify the block output when the simulation time reaches and exceeds the Step time parameter.

Programmatic Use
Block Parameter: After
Type: character vector
Values: '1' | real- or complex-valued scalar, vector, or matrix
Default: '1'

Sample time — Sample rate
0 (default) | scalar

Specify the sample rate of step. See “Specify Sample Time” for more information.
Programmatic Use
Block Parameter: SampleTime
Type: character vector
Values: scalar
Default: '0'

Interpret vector parameters as 1-D — Treat vectors as 1-D
on (default) | off

Select this check box to output a vector of length N if the Constant value parameter evaluates to an N-element row or column vector.

• When you select this check box, the block outputs a vector of length N if the Constant value parameter evaluates to an N-element row or column vector. For example, the block outputs a matrix of dimension 1-by-N or N-by-1.

• When you clear this check box, the block does not output a vector of length N if the Constant value parameter evaluates to an N-element row or column vector.

Programmatic Use
Block Parameter: VectorParams1D
Type: character vector
Values: 'on' | 'off'
Default: 'on'

Enable zero-crossing detection — Enable zero-crossing detection
on (default) | Boolean

Select to enable zero-crossing detection. For more information, see “Zero-Crossing Detection” in the Simulink documentation.

Programmatic Use
Block Parameter: ZeroCross
Type: character vector, string
Values: 'off' | 'on'
Default: 'on'

Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>double</th>
</tr>
</thead>
</table>

1-1987
<table>
<thead>
<tr>
<th>Feature</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>no</td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>yes</td>
</tr>
</tbody>
</table>

### Extended Capabilities

#### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Not recommended for production code.

### See Also

Ramp | Repeating Sequence Stair | Signal Editor

### Topics

“Signal Basics”

Introduced before R2006a
Stop Simulation

Stop simulation when input is nonzero

Library: Simulink / Sinks
HDL Coder / Sinks

Description

The Stop Simulation block stops the simulation when the input is nonzero. The simulation completes the current time step before terminating. If the block input is a vector, any nonzero vector element causes the simulation to stop.

When you use the Stop Simulation block in a For Iterator subsystem, the stop action occurs after execution of all iterations in the subsystem during a time step. The stop action does not interrupt execution until the start of the next time step.

You cannot use the Stop Simulation block to pause the simulation. To create a block that pauses the simulation, see “Pause Simulation Using Assertion Blocks”.

Ports

Input

Port_1 — Input signal
scalar | vector | matrix

Stop simulation when input signal is nonzero. This port accepts real signals of double or Boolean data types.

Data Types: double | Boolean
Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>double</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
</tr>
</tbody>
</table>

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

Not recommended for production code. Relates to resource limits and restrictions on speed and memory often found in embedded systems. Generated code can contain dynamic allocation and freeing of memory, recursion, additional memory overhead, and widely varying execution times. While the code is functionally valid and acceptable in resource-rich environments, smaller embedded targets often cannot support such code. Usually, blocks evolve toward being suitable for production code. Thus, blocks suitable for production code remain suitable.

Generated code stops executing when the stop condition is true.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

This block can be used to stop simulation when used with subsystems that generates code, but is not included in the hardware implementation.
PLC Code Generation
Generate Structured Text code using Simulink® PLC Coder™.

See Also

Topics
“Simulate a Model Interactively”

Introduced before R2006a
String Compare

Compare two input strings
Library: Simulink / String

Description

String Compare compares two strings. To see if two strings are identical, use this block. You can specify if the match is case sensitive and how much of the string to compare.

Ports

Input

Port_1 — First string to compare
scalar

First string to compare, specified as a scalar.
Data Types: string

Port_2 — Second string to compare
scalar

Second string to compare, specified as a scalar.
Data Types: string

Output

Port_1 — True or false result
scalar

True or false result, specified as a scalar:
• 1 — Match.
• 0 — No match.

Data Types: Boolean

Parameters

Case sensitive — Case sensitivity for string comparison
on (default) | off

Case sensitivity for string comparison:

✓ on
  Consider string case when comparing strings.

☐ off
  Do not consider string case when comparing strings.

Compare Option — Amount of characters to compare
Entire string (default) | First N characters

Amount of string to compare:

• Entire string — Compare both entire strings.
• First N characters — Compare the first N characters of both strings.

Dependencies

Setting this parameter to First N characters enables the Number of characters parameter.

Number of characters — Number of characters to compare
1 (default) | scalar

Number of characters to compare

Dependencies

This parameter is enabled when the Compare Option parameter is set to First N characters.
Data Types: double

Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>string</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
</tr>
</tbody>
</table>

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also
ASCII to String | Compose String | Scan String | String Concatenate | String Constant | String Find | String Length | String to ASCII | String to Double | String to Enum | String to Single | Substring | To String

Topics
“Parse NMEA GPS Text Message”
“Simulink Strings”

Introduced in R2018a
String Concatenate

Concatenate input strings to form one output string

**Library:** Simulink / String

Description

The String Concatenate block concatenates multiple input strings, in order of their input, to form one output string. Use this block if you want to combine multiple strings into a single string.

Ports

**Input**

*Port_1 — First input string*

scalar

First input string, specified as a scalar.

*Data Types: string*

*Port_2 — Second input string*

scalar

Second input string, specified as a scalar.

*Data Types: string*

**Output**

*Port_1 — Concatenated string*

scalar
Concatenated string, specified as a scalar.

Data Types: string

Parameters

Number of Inputs — Number of input strings
scalar

Number of input strings to concatenate, specified as a scalar. You can specify from 2 to 512 input ports.

Output data type — Output data type
string (default) | scalar

Output data type, specified using the string data type to specify a string with no maximum length.

To specify a string data type with a maximum length, specify stringtype(N). For example, stringtype(128) creates a string data type with a maximum length of 128 characters.

Click the Show data type assistant button to display the Data Type Assistant, which helps you set the data type attributes. See “Specify Data Types Using Data Type Assistant” in the Simulink User's Guide for more information.

Mode — Category of data
stringtype(128) (default) | scalar

Use the stringtype function, for example, stringtype(128).

Dependency

Clicking the Show data type assistant button enables this parameter.
Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>string</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>yes</td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>no</td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
</tr>
</tbody>
</table>

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also
ASCII to String | Compose String | Scan String | String Compare | String Constant | String Find | String Length | String to ASCII | String to Double | String to Enum | String to Single | Substring | To String

Topics
“Simulink Strings”

Introduced in R2018a
**String Constant**

Output specified string  
**Library:** Simulink / String

**Description**

The String Constant block outputs a string specified by the `String` parameter. Use this block when you want a constant whose type is string.

**Ports**

**Output**

*Port_1 — Output string*

*scalar*

Output string, specified as a scalar.  
Data Types: `string`

**Parameters**

*String — Input string*

"Hello!" (default) | scalar

Input string, specified as a scalar.

*Output data type — Output data type*

string (default) | scalar
Output data type, specified using the string data type to specify a string with no maximum length.

To specify a string data type with a maximum length, specify `stringtype(N)`. For example, `stringtype(31)` creates a string data type with a maximum length of 31 characters.

Click the **Show data type assistant** button to display the **Data Type Assistant**, which helps you set the data type attributes. See “Specify Data Types Using Data Type Assistant” in the Simulink User's Guide for more information.

### Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>string</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>no</td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
</tr>
</tbody>
</table>

### Extended Capabilities

**C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.
See Also
ASCII to String | Compose String | Scan String | String Compare | String Concatenate | String Find | String Length | String to ASCII | String to Double | String to Enum | String to Single | Substring | To String

Topics
“Convert String to ASCII and Back to String”
“Simulink Strings”

Introduced in R2018a
String Find

Return index of first occurrence of pattern string

Library: Simulink / String

Description

The String Find block returns the index of the first occurrence of the pattern string `sub` in the text string `str`.

Ports

Input

`str` — String in which to find pattern

scalar

String in which to find pattern (`sub`), specified as a scalar.

Data Types: `string`

`sub` — Pattern

scalar

Pattern to be found in string (`str`), specified as a scalar.

Data Types: `string`

Output

`idx` — Position index of found pattern

scalar

Position index of the found pattern, specified as a positive integer scalar.
• If the block does not find the pattern, it returns -1.
• If the \texttt{sub} parameter is empty (""), the block returns 1, indicating that it matched the beginning of the searched string.

Data Types: int32

### Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>integer</th>
<th>string</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
</tr>
</tbody>
</table>

### Extended Capabilities

**C/C++ Code Generation**
Generate C and C++ code using Simulink® Coder™.

### See Also

Compose String | Scan String | String Compare | String Concatenate | String Constant | String Find | String Length | String to ASCII | String to Double | String to Enum | String to Single | Substring | To String

### Topics

“Find Patterns in Strings”
“Simulink Strings”
Introduced in R2018a
String Length

Output number of characters in input string

Library: Simulink / String

Description

The String Length block outputs the number of characters in the input string. For example, you can use the String Length block to move focus of attention to a particular location in a string.

Ports

Input

Port_1 — Input string
scalar

Input string, specified as a scalar.
Data Types: string

Output

Port_1 — Number of characters
scalar

Number of characters in the input string, specified as a scalar.
Data Types: uint16 | uint32
## Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>integer</th>
<th>string</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
</tr>
</tbody>
</table>

## Extended Capabilities

### C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

### See Also
Compose String | Scan String | String Compare | String Concatenate | String Constant | String Find | String to ASCII | String to Double | String to Enum | String to Single | Substring | To String

### Topics
“Get Text Following a Keyword”
“Simulink Strings”

### Introduced in R2018a
String to ASCII

Convert string signal to uint8 vector

Library: Simulink / String

Description

The String To ASCII block converts a string signal to a uint8 vector. The block converts each character in the string to its corresponding ASCII value. For example, the block converts the input string "Hello" to [72 101 108 108 111].

Ports

Input

Port_1 — Input string signal
scalar

Input string signal, specified as a scalar.
Data Types: string

Output

Port_1 — Converted uint8 vector signal
vector

Converted uint8 vector signal of ASCII characters from input string signal, specified as a vector. The block converts each element in the string into its ASCII character equivalent and outputs the ASCII equivalents as a vector. If there are fewer characters than the maximum length, the block fills the remaining space with zeros at simulation. At code generation, the block fills the remaining space with null characters.
Data Types: uint8
Parameters

**Output vector size — Size of output vector**
31 (default) | scalar

Size of output string vector, specified as a scalar.

Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>integer</th>
<th>string</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
</tr>
</tbody>
</table>

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also
ASCII to String | Compose String | Scan String | String Compare | String Concatenate | String Constant | String Find | String Length | String to Double | String to Enum | String to Single | Substring | To String

Topics
“Convert String to ASCII and Back to String”
“Simulink Strings”

*Introduced in R2018a*
String to Double

Convert string signal to double signal

Library: Simulink / String

Description

Scan String scans an input string and converts it to signals per the format specified by the Format parameter. The block converts values to their decimal (base 10) representation and outputs the results as numeric or string signals. For example, if the Format parameter is set to "%s is %f.", the block outputs two parts, a string signal and a single signal. If the input is the string "Pi is 3.14", the two outputs are "Pi" and "3.14".

The Scan String, String to Double, and String to Single blocks are identical blocks. When configured for String to Double, the block converts the input string signal to a double numerical output. When configured for String to Single, the block converts the input string signal to a single numerical output.

For code generation, configure models that contain this block for non-finite number support by selecting the Configuration Parameters > Code Generation > Interface > Support non-finite numbers check box.

Ports

Input

Port_1 — Input string

scalar

Input string, specified as a scalar.

Data Types: string
Output

d — Output data whose format matches %d format
scalar

Output data whose format matches specified format, defined as a scalar. Total maximum number of outputs is 128.

If the block cannot match an input string to a format operator specified in Format, it returns a warning and outputs an appropriate value (0 or "") for each unmatched format operator.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus

f — Output data whose format matches %f format
scalar

Output data whose format matches the %f format, specified as a scalar. Total maximum number of outputs is 128.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus

Port N — Output data whose format matches N format
scalar

Output data whose format matches N format, specified as a scalar. Total maximum number of outputs is 128.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus

Parameters

Format — Format operator for input
"%lf" (default) | scalar

Format operator for input, specified as a scalar. If the block cannot match the input string with the specified format, it returns 0. The return of 0 differs from the sscanf function return, which is an empty matrix if the function cannot match the input with the specified format.
- For the String to Double block, this parameter has a default value of `%lf`.
- For the String to Single block, this parameter has a default value of `%f`.

For more information about acceptable format operators, see the Algorithms section.

### Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>double</th>
<th>integer</th>
<th>single</th>
<th>string</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Algorithms

The Scan String block uses this format specifier prototype:

```
%[width][length]specifier
```

### Numeric Fields

This table lists available conversion specifiers to convert text to numeric outputs. The block converts values to their decimal (base 10) representation.

<table>
<thead>
<tr>
<th>Output Port Data Type</th>
<th>ConversionSpecifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer, signed</td>
<td><code>%d</code></td>
<td>Base 10</td>
</tr>
<tr>
<td>Integer, unsigned</td>
<td><code>%u</code></td>
<td>Base 10</td>
</tr>
<tr>
<td>Output Port Data Type</td>
<td>Conversion Specifier</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------------</td>
<td>----------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Floating-point number</td>
<td>%f, %e, or %g</td>
<td>Floating-point values. Input fields can contain NaN (case sensitive). Input fields that represents floating-point numbers can include leading + or - symbols and exponential notation using e or E. The conversion specifiers %f, %e, and %g all treat input fields the same way.</td>
</tr>
</tbody>
</table>

**Character Fields**

This table lists available conversion specifiers to convert text so that the output is a character array.

<table>
<thead>
<tr>
<th>Character Field Type</th>
<th>Conversion Specifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>String scalar</td>
<td>%s</td>
<td>Read the text until the block encounters whitespace.</td>
</tr>
<tr>
<td></td>
<td>%c</td>
<td>Read any single character, including whitespace. To read multiple characters at a time, specify field width. For example, %10c reads 10 characters at a time.</td>
</tr>
<tr>
<td>Pattern-matching</td>
<td>%[...]</td>
<td>Read only characters in the brackets up to the first nonmatching character or whitespace.</td>
</tr>
<tr>
<td></td>
<td>%[^...]</td>
<td>Read any characters not in the brackets up to the first matching character or whitespace.</td>
</tr>
<tr>
<td></td>
<td>Example: %[mus] reads 'summer' as 'summ'.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Example: %[^m] reads 'summer' as 'su'.</td>
<td></td>
</tr>
</tbody>
</table>
Optional Operators

- **Field Width** — To specify the maximum number of digits or text characters to read at a time, insert a number after the percent character. For example, `%10s` reads up to 10 characters at a time, including whitespace. `%4f` reads up to four digits at a time, including the decimal point.

- **Literal Text to Ignore** — This block must match the specified text immediately before or after the conversion specifier.

  Example: `Hello%s` reads "Hello!" as "o!".

Length Specifiers

The Scan String block supports the `h` and `l` length subspecifiers. These specifiers can change according to the **Configuration Parameters > Hardware Implementation > Number of bits** settings.

<table>
<thead>
<tr>
<th>Length</th>
<th>i</th>
<th>u</th>
<th>f e g</th>
<th>s c</th>
</tr>
</thead>
<tbody>
<tr>
<td>No length specifier</td>
<td>int</td>
<td>unsigned int</td>
<td>single</td>
<td>string</td>
</tr>
<tr>
<td>h</td>
<td>short</td>
<td>unsigned short</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>l</td>
<td>long</td>
<td>unsigned long</td>
<td>double</td>
<td>—</td>
</tr>
</tbody>
</table>

Notes for Specifiers that Specify Integer Data Types (`d, u`)

- Target `int`, `long`, `short` type sizes are controlled by settings in the **Configuration Parameters > Hardware Implementation** pane. For example, if the target `int` is 32 bits and the specifier is `%u`, then the expected input type will be `uint32`. For this example, the Scan String block requires that the output type be exactly `int32`. It cannot be any other data type.
Extended Capabilities

**C/C++ Code Generation**
Generate C and C++ code using Simulink® Coder™.

**See Also**
ASCII to String | Compose String | Scan String | String Compare | String Concatenate | String Constant | String Find | String Length | String to ASCII | String to Enum | String to Single | Substring | To String | `sscanf`

**Topics**
“Display and Extract Coordinate Data”
“Simulink Strings”

**Introduced in R2018a**
String to Enum

Input string signal to enumerated signal

Library: Simulink / String

Description

The String To Enum block converts the input string signal to an enumerated signal. To use this block, create an enumeration class in the current folder and use that class name in the Output data type parameter.

Ports

Input

Port_1 — Input string signal
scalar

Input string signal, specified as a scalar.
Data Types: string

Output

Output 1 — Enumerated number
scalar

Enumerated number associated with the input string, specified as a scalar.
Data Types: enumerated
Parameters

Output data type — Output data type
SlDemoSign (default)|<data type expression>

Use a data type object, for example, Simulink.IntEnumType.

Click the Show data type assistant button to display the Data Type Assistant, which helps you set the data type attributes. See “Specify Data Types Using Data Type Assistant” in the Simulink User's Guide for more information.

Mode — Category of data
Enumerated (default)|<data type expression>

Use a data type object, for example, Simulink.IntEnumType.

• Enumerated — Enumerated data class object.
• <data type expression> — Expressions that evaluate to data types. Selecting Expression enables a second menu/text box to the right, where you can enter the expression.

Dependency

Clicking the Show data type assistant button enables this parameter.

Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>enumerated</th>
<th>string</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
</tr>
</tbody>
</table>
Extended Capabilities

**C/C++ Code Generation**
Generate C and C++ code using Simulink® Coder™.

**See Also**
ASCII to String | Compose String | Scan String | String Compare | String Concatenate | String Constant | String Find | String Length | String To ASCII | String to Double | String to Single | Substring | To String

**Topics**
“Convert String to Enumerated Data Type”
“Simulink Strings”

**Introduced in R2018a**
String to Single

Convert string signal to single signal

Library: Simulink / String

Description

Scan String scans an input string and converts it to signals per the format specified by the **Format** parameter. The block converts values to their decimal (base 10) representation and outputs the results as numeric or string signals. For example, if the **Format** parameter is set to "%s is %f.", the block outputs two parts, a string signal and a single signal. If the input is the string "Pi is 3.14", the two outputs are "Pi" and "3.14".

The Scan String, String to Double, and String to Single blocks are identical blocks. When configured for String to Double, the block converts the input string signal to a double numerical output. When configured for String to Single, the block converts the input string signal to a single numerical output.

For code generation, configure models that contain this block for non-finite number support by selecting the **Configuration Parameters > Code Generation > Interface > Support non-finite numbers** check box.

Ports

Input

**Port_1 — Input string**

scalar

Input string, specified as a scalar.

Data Types: string
Output

d — Output data whose format matches %d format
scalar

Output data whose format matches specified format, defined as a scalar. Total maximum number of outputs is 128.

If the block cannot match an input string to a format operator specified in Format, it returns a warning and outputs an appropriate value (0 or "") for each unmatched format operator.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus

f — Output data whose format matches %f format
scalar

Output data whose format matches the %f format, specified as a scalar. Total maximum number of outputs is 128.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus

Port_N — Output data whose format matches N format
scalar

Output data whose format matches $N$ format, specified as a scalar. Total maximum number of outputs is 128.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus

Parameters

Format — Format operator for input
"%f" (default) | scalar

Format operator for input, specified as a scalar. If the block cannot match the input string with the specified format, it returns 0. The return of 0 differs from the sscanf function return, which is an empty matrix if the function cannot match the input with the specified format.
For the String to Double block, this parameter has a default value of `%lf`.
For the String to Single block, this parameter has a default value of `%f`.

For more information about acceptable format operators, see the Algorithms section.

**Block Characteristics**

<table>
<thead>
<tr>
<th>Data Types</th>
<th>double</th>
<th>integer</th>
<th>single</th>
<th>string</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Algorithms**

The Scan String block uses this format specifier prototype:

`%[width][length]specifier`

**Numeric Fields**

This table lists available conversion specifiers to convert text to numeric outputs. The block converts values to their decimal (base 10) representation.

<table>
<thead>
<tr>
<th>Output Port Data Type</th>
<th>Conversion Specifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer, signed</td>
<td><code>%d</code></td>
<td>Base 10</td>
</tr>
<tr>
<td>Integer, unsigned</td>
<td><code>%u</code></td>
<td>Base 10</td>
</tr>
</tbody>
</table>
### Output Port Data Type

<table>
<thead>
<tr>
<th>Output Port Data Type</th>
<th>Conversion Specifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Floating-point number</td>
<td>%f, %e, or %g</td>
<td>Floating-point values. Input fields can contain NaN (case sensitive). Input fields that represents floating-point numbers can include leading + or - symbols and exponential notation using e or E. The conversion specifiers %f, %e, and %g all treat input fields the same way.</td>
</tr>
</tbody>
</table>

### Character Fields

This table lists available conversion specifiers to convert text so that the output is a character array.

<table>
<thead>
<tr>
<th>Character Field Type</th>
<th>Conversion Specifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>String scalar</td>
<td>%s</td>
<td>Read the text until the block encounters whitespace.</td>
</tr>
<tr>
<td></td>
<td>%c</td>
<td>Read any single character, including whitespace. To read multiple characters at a time, specify field width. For example, %10c reads 10 characters at a time.</td>
</tr>
<tr>
<td>Pattern-matching</td>
<td>%[...]</td>
<td>Read only characters in the brackets up to the first nonmatching character or whitespace.</td>
</tr>
<tr>
<td></td>
<td>%[^...]</td>
<td>Read any characters not in the brackets up to the first matching character or whitespace.</td>
</tr>
</tbody>
</table>

Example: %[mus] reads 'summer' as 'summ'.

Example: %[^m] reads 'summer' as 'su'.
Optional Operators

- Field Width — To specify the maximum number of digits or text characters to read at a time, insert a number after the percent character. For example, %10s reads up to 10 characters at a time, including whitespace. %4f reads up to four digits at a time, including the decimal point.
- Literal Text to Ignore — This block must match the specified text immediately before or after the conversion specifier.

Example: Hell%s reads "Hello!" as "o!".

Length Specifiers

The Scan String block supports the h and l length subspecifiers. These specifiers can change according to the Configuration Parameters > Hardware Implementation > Number of bits settings.

<table>
<thead>
<tr>
<th>Length</th>
<th>i</th>
<th>u</th>
<th>f e g</th>
<th>s c</th>
</tr>
</thead>
<tbody>
<tr>
<td>No length specifier</td>
<td>int</td>
<td>unsigned int</td>
<td>single</td>
<td>string</td>
</tr>
<tr>
<td>h</td>
<td>short</td>
<td>unsigned short</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>l</td>
<td>long</td>
<td>unsigned long</td>
<td>double</td>
<td>—</td>
</tr>
</tbody>
</table>

Notes for Specifiers that Specify Integer Data Types (d, u)

- Target int, long, short type sizes are controlled by settings in the Configuration Parameters > Hardware Implementation pane. For example, if the target int is 32 bits and the specifier is %u, then the expected input type will be uint32. For this example, the Scan String block requires that the output type be exactly int32. It cannot be any other data type.
Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also
ASCII to String | Compose String | Scan String | String Compare | String Concatenate | String Constant | String Find | String Length | String to ASCII | String to Double | String to Enum | Substring | To String | sscanf

Topics
“Display and Extract Coordinate Data”
“Simulink Strings”

Introduced in R2018a
Substring

Extract substring from input string signal

**Library:** Simulink / String

**Description**

The Substring block extracts a substring from the input string signal. The block extracts the substring starting from the letter corresponding to `idx` and includes a `len` number of characters starting at `idx`. For example, if the input string is "hello 123", input `idx` is 1, and input `len` is 5, the output is "hello". The block extracts a substring starting at 1 and the next 4 characters for a total of 5 characters (hello).

**Ports**

**Input**

- **str** — Input string signal
  
scalar
  
Input string signal, specified as a string.
  
Data Types: `string`

- **idx** — Start of string to extract
  
scalar
  
Start of string to extract, specified as a positive scalar integer.
  
Data Types: `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32`

- **len** — Length of string to extract
  
scalar
Length of string to extract, specified as a scalar. If \textbf{len} causes the substring to extend beyond the end of the string, the output signal contains fewer than \textbf{len} characters.

Data Types: \texttt{uint8} | \texttt{uint16} | \texttt{uint32}

\textbf{Output}

\textit{sub — Extracted string}

\texttt{scalar}

Extracted string, specified as a scalar.

Data Types: \texttt{string}

\textbf{Parameters}

\texttt{Inherit maximum length from input — Use same maximum length as input string}

on (default) | off

Use same maximum length as the input string source block.

\checkmark on

Use same maximum length. The substring includes the characters starting from the character at \texttt{idx} to the end of the string.

\square off

Do not use same maximum length.

\texttt{Output string from 'idx' to end — Extract string from idx to end}

off (default) | on

Extract string from \texttt{idx} to end of input string.

\checkmark on

Extract string from \texttt{idx} to end of input string.

\square off

Do not extract string from \texttt{idx} to end of input string.
Dependencies

Selecting this parameter removes the third input port.

Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>integer</th>
<th>string</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
</tr>
</tbody>
</table>

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

ASCII to String | Compose String | Scan String | String Compare | String Concatenate | String Constant | String Find | String Length | String to ASCII | String to Double | String to Enum | String to Single | To String

Topics

“Extract a String”
“Simulink Strings”
Introduced in R2018a
**Subsystem, Atomic Subsystem, CodeReuse Subsystem**

Group blocks to create model hierarchy

**Library:** Simulink / Ports & Subsystems

---

**Description**

A Subsystem block contains a subset of blocks within a model or system. The Subsystem block can represent a virtual subsystem or a nonvirtual subsystem.

- **Nonvirtual subsystem** – Control when the contents of the subsystem are evaluated as a single unit (atomic execution). Create conditionally executed subsystems that run only when an event occurs on a triggering, function-call, action, or enabling input (see “Conditionally Executed Subsystems and Models”).

- **Virtual subsystem** – Subsystem is neither conditionally nor atomically executed. Virtual subsystems do not have checksums. To determine if a subsystem is virtual, use the `get_param` function for the Boolean block parameter `IsSubsystemVirtual`.

An atomic subsystem is a Subsystem block with the block parameter `Treat as atomic unit` selected.

A codereuse subsystem is a Subsystem block with the parameter `Treat as atomic unit` selected and the parameter `Function packaging` set to `Reusable function`, specifying the function code generation format for the subsystem.

A referenced subsystem saves the content of a subsystem in a separate file and reference it using a Subsystem Reference block. To reuse that subsystem, add multiple Subsystem Reference blocks that reference that same subsystem file. A referenced subsystem behaves identical to a regular subsystem during run-time.

To create a subsystem, do one of the following:

- Copy a Subsystem block from the Ports & Subsystems library into your model. Then add blocks to the subsystem by opening the Subsystem block and copying blocks into it.
• Select all blocks and lines that make up the subsystem, and then in the Multiple tab, click Create Subsystem. Simulink replaces the blocks with a Subsystem block, along with the necessary Inport and Outport blocks to reflect signals entering and leaving the subsystem.

The number of input ports drawn on the Subsystem block icon corresponds to the number of Inport blocks in the subsystem. Similarly, the number of output ports drawn on the block corresponds to the number of Outport blocks in the subsystem.

The Subsystem block supports signal label propagation through subsystem Inport and Outport blocks.

### Ports

#### Input

**In — Signal input to a subsystem block**

| scalar | vector | matrix |

Placing an Inport block in a subsystem block adds an external input port to the block. The port label matches the name of the Inport block.

Use Inport blocks to get signals from the local environment.

**Data Types:** single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus

#### Output

**Out — Signal output from a subsystem**

| scalar | vector | matrix |

Placing an Outport block in a subsystem block adds an output port from the block. The port label on the subsystem block is the name of the Outport block.

Use Outport blocks to send signals to the local environment.

**Data Types:** single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus
Parameters

Parameters on the Code Generation tab require a Simulink Coder or Embedded Coder license.

**Content Preview — Display representation of block contents on the block**

'off' (default) | 'on'

Display a representation of block contents on the block. For more information, see “Preview Content of Model Components”.

**Programmatic Use**

**Parameter**: ContentPreviewEnabled

**Type**: character vector

**Value**: 'on' | 'off'

**Default**: 'on'

Main

**Show port labels — Select how to display port labels**

FromPortIcon (default) | FromPortBlockName | SignalName

Select how to display port labels on the Subsystem block icon.

none

Do not display port labels.

FromPortIcon

If the corresponding port icon displays a signal name, display the signal name on the Subsystem block. Otherwise, display the port block name.

FromPortBlockName

Display the name of the corresponding port block on the Subsystem block.

SignalName

If a signal name exists, display the name of the signal connected to the port on the Subsystem block. Otherwise, display the name of the corresponding port block.

**Programmatic Use**

**Parameter**: ShowPortLabels

**Type**: character vector
Value: 'FromPortIcon' | 'FromPortBlockName' | 'SignalName'
Default: 'FromPortIcon'

**Read/Write permissions — Select access to contents of subsystem**
ReadWrite (default) | ReadOnly | NoReadOrWrite

Control user access to the contents of the subsystem.

**ReadWrite**
Enable opening and modification of subsystem contents.

**ReadOnly**
Enable opening but not modification of the subsystem. If the subsystem resides in a block library, you can create and open links to the subsystem and can make and modify local copies of the subsystem but cannot change the permissions or modify the contents of the original library instance.

**NoReadOrWrite**
Disable opening or modification of subsystem. If the subsystem resides in a library, you can create links to the subsystem in a model but cannot open, modify, change permissions, or create local copies of the subsystem.

**Programmatic Use**
**Parameter:** Permissions
**Type:** character vector
**Value:** 'ReadWrite' | 'ReadOnly' | 'NoReadOrWrite'
**Default:** 'ReadWrite'

**Name of error callback function — Specify name of function**
' ' (default) | function name

Enter name of a function to be called if an error occurs while Simulink is executing the subsystem.

Simulink passes two arguments to the function: the handle of the subsystem and a character vector that specifies the error type. If no function is specified, Simulink displays a generic error message if executing the subsystem causes an error.

**Programmatic Use**
**Parameter:** ErrorFcn
**Type:** character vector
**Value:** '' | '<function name>'
Default: ''

Permit hierarchical resolution — Select how to resolve workspace variable names
All (default) | ExplicitOnly | None

Select whether to resolve names of workspace variables referenced by this subsystem.


All
- Resolve all names of workspace variables used by this subsystem, including those used to specify block parameter values and Simulink data objects (for example, Simulink.Signal objects).

ExplicitOnly
- Resolve only names of workspace variables used to specify block parameter values, data store memory (where no block exists), signals, and states marked as “must resolve”.

None
- Do not resolve any workspace variable names.

Programmatic Use
Parameter: PermitHierarchicalResolution
Type: character vector
Value: 'All' | 'ExplicitOnly' | 'None'
Default: 'All'

Treat as atomic unit — Control execution of a subsystem as one unit
off (default) | on

Causes Simulink to treat the subsystem as a unit when determining the execution order of block methods.

\[ \text{\square} \text{ off} \]

- Treat all blocks in the subsystem as being at the same level in the model hierarchy as the subsystem when determining block method execution order. This can cause execution of methods of blocks in the subsystem to be interleaved with execution of methods of blocks outside the subsystem.
on

Treat the subsystem as a unit when determining the execution order of block methods. For example, when it needs to compute the output of the subsystem, Simulink invokes the output methods of all the blocks in the subsystem before invoking the output methods of other blocks at the same level as the subsystem block.

**Dependency**

Selecting this parameter enables the **Minimize algebraic loop occurrences**, **Sample time**, **Schedule As** and **Function packaging** parameters. Using **Function packaging** requires a Simulink Coder license.

**Programmatic Use**

- **Parameter**: TreatAsAtomicUnit
- **Type**: character vector
- **Value**: 'off' | 'on'
- **Default**: 'off'

**Minimize algebraic loop occurrences — Control elimination of algebraic loops**

- **off** (default) | **on**

Try to eliminate any artificial algebraic loops that include the atomic subsystem.

- **off**

Do not try to eliminate any artificial algebraic loops that include the atomic subsystem.

- **on**

Try to eliminate any artificial algebraic loops that include the atomic subsystem.

**Dependency**

To enable this parameter, select the **Treat as atomic unit** parameter.

**Programmatic Use**

- **Parameter**: MinAlgLoopOccurrences
- **Type**: character vector
- **Value**: 'off' | 'on'
- **Default**: 'off'
**Schedule As — Specify type of scheduling**

Sample Time (default) | Periodic Partition | Aperiodic Partition

Specify how to schedule the subsystem.

**Sample time**

Specify whether all blocks in this subsystem must run at the same rate or can run at different rates.

**Periodic Partition**

Schedule the subsystem as a periodic partition. Specify a partition name and a sample time corresponding to the rate at which the partition runs.

**Aperiodic Partition**

Schedule the subsystem as an aperiodic partition. Specify a partition name.

**Dependency**

To enable this parameter, select the **Treat as atomic unit** parameter.

**Programmatic Use**

**Parameter:** ScheduleAs  
**Type:** character vector  
**Value:** 'SampleTime' | 'DiscretePartition' | 'UnconstrainedPartition'  
**Default:** 'SampleTime'

**Partition Name — Specify name of the partition**

**Character Vector**

Specify name of the partition for the subsystem.

**Dependency**

To enable this parameter, select the **Treat as atomic unit** parameter, and **Schedule As** to be Discrete Partition or Unconstrained partition.

**Programmatic Use**

**Parameter:** PartitionName  
**Type:** character vector  
**Value:** character vector  
**Default:** character vector

**Sample time — Specify time interval**

-1 (default) | [Ts 0]
Specify whether all blocks in this subsystem must run at the same rate or can run at different rates.

- If the blocks in the subsystem can run at different rates, specify the subsystem's sample time as inherited (-1).
- If all blocks must run at the same rate, specify the sample time corresponding to this rate as the value of the Sample time parameter.
- If any of the blocks in the subsystem specify a different sample time (other than -1 or inf), Simulink displays an error message when you update or simulate the model. For example, suppose all the blocks in the subsystem must run 5 times a second. To ensure this, specify the sample time of the subsystem as 0.2. In this example, if any of the blocks in the subsystem specify a sample time other than 0.2, -1, or inf, Simulink displays an error when you update or simulate the model.

-1
   Specify inherited sample time. Use this sample time if the blocks in the subsystem can run at different rates.

[Ts 0]
   Specify periodic sample time.

**Dependency**

To enable this parameter, select the Treat as atomic unit parameter.

**Programmatic Use**

Parameter: SystemSampleTime
Type: character vector
Value: '-1' | '[Ts 0]'
Default: '-1'

**Propagate execution context across subsystem boundary — Control execution across block boundary**

off (default) | on

Enable execution context propagation across the boundary of this subsystem.

off

Do not enable execution context propagation across this subsystem's boundary.
on

Enable execution context propagation across this subsystem's boundary.

**Dependency**

Enable this parameter by adding an Enable port or Trigger port block to the Subsystem block.

**Programmatic Use**

**Parameter:** PropExecContextOutsideSubsystem  
**Type:** character vector  
**Value:** 'off' | 'on'  
**Default:** 'off'

**Variant control — Specify variant control (condition) expression**

Variant (default) | logical expression

Specify variant control (condition) expression that executes a variant Simulink Function block when the expression evaluates to true.

See also `Simulink.Variant`

**Variant**

Default name for a logical (Boolean) expression.

**logical expression**

A logical (Boolean) expression or a `Simulink.Variant` object representing a logical expression.

The function is activated when the expression evaluates to true.

If you want to generate code for your model, define the variables in the expression as `Simulink.Parameter` objects.

**Dependency**

Enable this parameter by adding a Subsystem block inside a Variant Subsystem block.

**Programmatic Use**

**Block parameter:** VariantControl  
**Type:** character vector  
**Value:** 'Variant' | '<logical expression>'
Default: 'Variant'

**Treat as grouped when propagating variant conditions — Control treating subsystem as unit**

*on* (default) | *off*

Causes Simulink to treat the subsystem as a unit when propagating variant conditions from Variant Source blocks or to Variant Sink blocks.

- **on**
  - Simulink treats the subsystem as a unit when propagating variant conditions from Variant Source blocks or to Variant Sink blocks. For example, when Simulink computes the variant condition of the subsystem, it propagates that condition to all the blocks in the subsystem.

- **off**
  - Simulink treats all blocks in the subsystem as being at the same level in the model hierarchy as the subsystem itself when determining their variant condition.

**Programmatic Use**

**Parameter:** TreatAsGroupedWhenPropagatingVariantConditions

**Type:** character vector  
**Value:** 'on' | 'off'

**Default:** 'on'

**Code Generation**

**Function packaging — Select code format**

*Auto* (default) | *Inline* | *Nonreusable function* | *Reusable function*

Select the code format to be generated for an atomic (nonvirtual) subsystem.

- **Auto**
  - Simulink Coder and Embedded Coder choose the optimal format for you based on the type and number of instances of the subsystem that exist in the model.

- **Inline**
  - Simulink Coder and Embedded Coder inline the subsystem unconditionally.

- **Nonreusable function**
  - If **Filename options** is set to *Auto*, Simulink Coder and Embedded Coder package separate functions in the model file. If **File name options** is set to **Use subsystem**
name, Use function name, or User specified using different file names, Simulink Coder and Embedded Coder package separate functions in separate files.

Subsystems with this setting generate functions that might have arguments depending on the “Function interface” on page 1-0 parameter setting. You can name the generated function and file using parameters “Function name” on page 1-0 and “File name (no extension)” on page 1-0. These functions are not reentrant.

Reusable function

Simulink Coder and Embedded Coder generate a function with arguments that allows reuse of subsystem code when a model includes multiple instances of the subsystem.

This option also generates a function with arguments that allows subsystem code to be reused in the generated code of a model reference hierarchy that includes multiple instances of a subsystem across referenced models. In this case, the subsystem must be in a library.

**Tips**

- When you want multiple instances of a subsystem to be represented as one reusable function, you can designate each one of them as Auto or as Reusable function. It is best to use one or the other, as using both creates two reusable functions, one for each designation. The outcomes of these choices differ only when reuse is not possible. Selecting Auto does not allow control of the function or file name for the subsystem code.

- The Reusable function and Auto options both try to determine if multiple instances of a subsystem exist and if the code can be reused. The difference between the options' behavior is that when reuse is not possible:
  
  - Auto yields inlined code, or if circumstances prohibit inlining, separate functions for each subsystem instance.
  - Reusable function yields a separate function with arguments for each subsystem instance in the model.

- If you select Reusable function while your generated code is under source control, set **File name options** to **Use subsystem name**, **Use function name**, or **User specified**. Otherwise, the names of your code files change whenever you modify your model, which prevents source control on your files.
Dependency

- This parameter requires Simulink Coder for code generation.
- To enable this parameter, select **Treat as atomic unit**.
- Setting this parameter to **Nonreusable function** or **Reusable function** enables the following parameters:
  - **Function name options**
  - **File name options**
  - Memory section for initialize/terminate functions (requires a license for Embedded Coder and an ERT-based system target file)
  - Memory section for execution functions (requires a license for Embedded Coder and an ERT-based system target file)
- Setting this parameter to **Nonreusable function** enables **Function with separate data** (requires a license for Embedded Coder and an ERT-based system target file).

Programmatic Use

**Parameter:** RTWSystemCode  
**Type:** character vector  
**Value:** 'Auto' | 'Inline' | 'Nonreusable function' | 'Reusable function'  
**Default:** 'Auto'

See also

- “Create a Subsystem”
- “Control Generation of Functions for Subsystems” (Simulink Coder)
- “Generate Code and Executables for Individual Subsystems” (Simulink Coder)
- “Inline Subsystem Code” (Simulink Coder)
- “Generate Subsystem Code as Separate Function and Files” (Simulink Coder)
- “Generate Reusable Code from Library Subsystems Shared Across Models” (Simulink Coder)
- “Generate Reusable Code from Library Subsystems Shared Across Models” (Simulink Coder)

**Function name options — Select how to name generated function**

Auto (default) | Use subsystem name | User specified

Select how Simulink Coder names the function it generates for the subsystem.
If you have an Embedded Coder license, you can control function names with options on the Configuration Parameter **Code Generation > Identifiers** pane.

**Auto**

Assign a unique function name using the default naming convention, `model_subsystem()`, where `model` is the name of the model and `subsystem` is the name of the subsystem (or that of an identical one when code is being reused).

If you select **Reusable function** for the **Function packaging** parameter and there are multiple instances of the reusable subsystem in a model reference hierarchy, in order to generate reusable code for the subsystem, **Function name options** must be set to **Auto**.

**Use subsystem name**

Use the subsystem name as the function name. By default, the function name uses the naming convention `model_subsystem`.

**Note** When a subsystem is in a library block and the subsystem parameter “Function packaging” on page 1-0 is set to **Reusable function**, if you set the **Use subsystem name** option, the code generator uses the name of the library block for the subsystem's function name and file name.

**User specified**

Enable the **Function name** field. Enter any legal C or C++ function name, which must be unique.

**Dependency**

- This parameter requires a Simulink Coder license.
- Setting Code generation function packaging to **Nonreusable function** or **Reusable function** enables this parameter.
- Setting this parameter to **User specified** enables the Code generation function name parameter.

**Programmatic Use**

**Parameter:** RTWFcnNameOpts  
**Type:** character vector  
**Value:** 'Auto' | 'Use subsystem name' | 'User specified'  
**Default:** 'Auto'
See also

For more information, see “Control Generation of Functions for Subsystems” (Simulink Coder).

**Function name — Specify function name**

```
'' (default) | function name
```

Specify a unique, valid C or C++ function name for subsystem code.

Use this parameter if you want to give the function a specific name instead of allowing the Simulink Coder code generator to assign its own autogenerated name or use the subsystem name. For more information, see “Control Generation of Functions for Subsystems” (Simulink Coder).

**Dependency**

- This parameter requires a Simulink Coder license.
- To enable this parameter, set the **Function name options** parameter to **User specified**.

**Programmatic Use**

Parameter: RTWFcnName  
Type: character vector  
Value: '' | '<function name>'  
Default: ''

**File name options — Specify how to name generated file**

Auto (default) | Use subsystem name | Use function name | User specified

Select how Simulink Coder names the separate file for the function it generates for the subsystem.

See “Block-Specific Parameters” on page 6-132 for the command-line information.

**Auto**

Depending on the configuration of the subsystem and how many instances are in the model, **Auto** yields different results:

- If the code generator does not generate a separate file for the subsystem, the subsystem code is generated within the code module generated from the subsystem's parent system. If the subsystem's parent is the model itself, the subsystem code is generated within *model.c* or *model.cpp*. 
• If you select Reusable function for the Function packaging parameter and your generated code is under source control, consider specifying a File name options value other than Auto. This prevents the generated file name from changing due to unrelated model modifications, which is problematic for using source control to manage configurations.

• If you select Reusable function for the Function packaging parameter and there are multiple instances of the reusable subsystem in a model reference hierarchy, in order to generate reusable code for the subsystem, File name options must be set to Auto.

Use subsystem name
The code generator generates a separate file, using the subsystem (or library block) name as the file name.

Note When File name options is set to Use subsystem name, the subsystem file name is mangled if the model contains Model blocks, or if a model reference target is being generated for the model. In these situations, the file name for the subsystem consists of the subsystem name prefixed by the model name.

Use function name
The code generator uses the function name specified by Function name options as the file name.

User specified
This option enables the File name (no extension) text entry field. The code generator uses the name you enter as the file name. Enter any file name, but do not include the .c or .cpp (or any other) extension. This file name need not be unique.

Note While a subsystem source file name need not be unique, you must avoid giving nonunique names that result in cyclic dependencies (for example, sys_a.h includes sys_b.h, sys_b.h includes sys_c.h, and sys_c.h includes sys_a.h).

Dependency
• This parameter requires a Simulink Coder license.
• To enable this parameter, set Function packaging to Nonreusable function or Reusable function.
• Setting this parameter to User specified enables the File name (no extension) parameter.

**Programmatic Use**
**Parameter:** RTWFileNameOpts  
**Type:** character vector  
**Value:** 'Auto' | 'Use subsystem name' | 'Use function name' | 'User specified'  
**Default:** 'Auto'

**File name (no extension) — Specify file name**
' ' (default) | file name

The file name that you specify does not have to be unique. However, avoid giving non-unique names that result in cyclic dependencies (for example, `sys_a.h` includes `sys_b.h`, `sys_b.h` includes `sys_c.h`, and `sys_c.h` includes `sys_a.h`).

For more information, see “Control Generation of Functions for Subsystems” (Simulink Coder).

**Dependency**

• This parameter requires a Simulink Coder license.
• To enable this parameter, set File name options to User specified.

**Programmatic Use**
**Parameter:** RTWFileName  
**Type:** character vector  
**Value:** ' ' | '<file name>'  
**Default:** '

**Function with separate data — Control code generation for subsystem**
off (default) | on

Generate subsystem function code in which the internal data for an atomic subsystem is separated from its parent model and is owned by the subsystem.

on

Do not generate subsystem function code in which the internal data for an atomic subsystem is separated from its parent model and is owned by the subsystem.
on

Generate subsystem function code in which the internal data for an atomic subsystem is separated from its parent model and is owned by the subsystem. The subsystem data structure is declared independently from the parent model data structures. A subsystem with separate data has its own block I/O and DWork data structure. As a result, the generated code for the subsystem is easier to trace and test. The data separation also tends to reduce the maximum size of global data structures throughout the model, because they are split into multiple data structures.

Dependency

- This parameter requires a license for Embedded Coder and an ERT-based system target file.
- To enable this parameter, set Function packaging to Nonreusable function.
- Selecting this parameter enables these parameters:
  - Memory section for constants
  - Memory section for internal data
  - Memory section for parameters

Programmatic Use

Parameter: FunctionWithSeparateData
Type: character vector
Value: 'off' | 'on'
Default: 'off'

See also

- See the Subsystem block reference page for more information.
- For details on how to generate modular function code for an atomic subsystem, see “Generate Modular Function Code for Nonvirtual Subsystems” (Embedded Coder).
- For details on how to apply memory sections to atomic subsystems, see “Override Default Memory Placement for Subsystem Functions and Data” (Embedded Coder).

Function interface — Select to use arguments with generate function

void_void (default) | Allow arguments

Select to use arguments with generated function.
Generate a function without arguments and pass data as global variables. For example:

```c
void subsystem_function(void)
```

**Allow arguments**

Generate a function that uses arguments instead of passing data as global variables. This specification reduces global RAM. It might reduce code size and improve execution speed, and allow the code generator to apply additional optimizations. For example:

```c
void subsystem_function(real_T rtu_In1, real_T rtu_In2, real_T *rty_Out1)
```

**Dependency**

- This parameter requires an Embedded Coder license and an ERT-based system target file.
- To enable this parameter, set **Function packaging** to **Nonreusable function**.

**Programmatic Use**

**Parameter:** FunctionInterfaceSpec  
**Type:** character vector  
**Value:** 'void_void' | 'Allow arguments'  
**Default:** 'void_void'

**See also**

- “Reduce Global Variables in Nonreusable Subsystem Functions” (Embedded Coder)  
- “Generate Modular Function Code for Nonvirtual Subsystems” (Embedded Coder)

**Memory section for initialize/terminate functions — Select how to apply memory sections**

**Inherit from model (default)**  
**Default**  
**The memory section of interest**

Select how Embedded Coder applies memory sections to the subsystem initialization and termination functions.

**Inherit from model**

- Apply the root model's memory sections to the subsystem's function code
Default
Do not apply memory sections to the subsystem's system code, overriding any model-level specification

The memory section of interest
Apply one of the model's memory sections to the subsystem

Tips
• The possible values vary depending on what (if any) package of memory sections you have set for the model's configuration. See “Control Data and Function Placement in Memory by Inserting Pragmas” (Embedded Coder) and “Model Configuration Parameters: Code Generation” (Simulink Coder).
• If you have not configured the model with a package, Inherit from model is the only value that appears. Otherwise, the list includes Default and all memory sections the model's package contains.
• These options can be useful for overriding the model's memory section settings for the given subsystem.

Dependency
• This parameter requires a license for Embedded Coder software and an ERT-based system target file.
• To enable this parameter, set Function packaging to Nonreusable function or Reusable function.

Programmatic Use
Parameter: RTWMemSecFuncInitTerm
Type: character vector
Value: 'Inherit from model' | 'Default' | 'The memory section of interest'
Default: 'Inherit from model'

See also
• See the Subsystem block reference page for more information.
• For details on how to apply memory sections to atomic subsystems, see “Override Default Memory Placement for Subsystem Functions and Data” (Embedded Coder).

Memory section for execution functions — Select how to apply memory sections
Inherit from model (default) | Default | The memory section of interest
Select how Embedded Coder applies memory sections to the subsystem's execution functions.

**Inherit from model**
- Apply the root model's memory sections to the subsystem's function code

**Default**
- Do not apply memory sections to the subsystem system code, overriding any model-level specification

**The memory section of interest**
- Apply one of the model's memory sections to the subsystem

**Tips**
- The possible values vary depending on what (if any) package of memory sections you have set for the model's configuration. See “Control Data and Function Placement in Memory by Inserting Pragmas” (Embedded Coder) and “Model Configuration Parameters: Code Generation” (Simulink Coder).
- If you have not configured the model with a package, Inherit from model is the only value that appears. Otherwise, the list includes Default and all memory sections the model's package contains.
- These options can be useful for overriding the model's memory section settings for the given subsystem.

**Dependency**
- This parameter requires a license for Embedded Coder software and an ERT-based system target file.
- To enable this parameter, set **Function packaging** to Nonreusable function or Reusable function.

**Programmatic Use**
- **Parameter**: RTWMemSecFuncExecute
- **Type**: character vector
- **Value**: 'Inherit from model' | 'Default' | 'The memory section of interest'
- **Default**: 'Inherit from model'

**See also**
- See the Subsystem block reference page for more information.
For details on how to apply memory sections to atomic subsystems, see “Override Default Memory Placement for Subsystem Functions and Data” (Embedded Coder).

**Memory section for constants — Select how to apply memory sections**

- Inherit from model (default)
- Default
- The memory section of interest

Select how Embedded Coder applies memory sections to the subsystem constants.

**Inherit from model**

- Apply the root model's memory sections to the subsystem's data

**Default**

- Not apply memory sections to the subsystem's data, overriding any model-level specification

**The memory section of interest**

- Apply one of the model's memory sections to the subsystem

**Tips**

- The memory section that you specify applies to the corresponding global data structures in the generated code. For basic information about the global data structures generated for atomic subsystems, see “Standard Data Structures in the Generated Code” (Simulink Coder).
- Can be useful for overriding the model's memory section settings for the given subsystem.
- The possible values vary depending on what (if any) package of memory sections you have set for the model's configuration. See “Control Data and Function Placement in Memory by Inserting Pragmas” (Embedded Coder).
- If you have not configured the model with a package, Inherit from model is the only value that appears. Otherwise, the list includes Default and all memory sections the model's package contains.

**Dependency**

- This parameter requires a license for Embedded Coder and an ERT-based system target file.
- To enable this parameter, set Function packaging to Nonreusable function and select the Function with separate data parameter
**Programmatic Use**

**Parameter:** RTWMemSecDataConstants  
**Type:** character vector  
**Value:** 'Inherit from model' | 'Default' | 'The memory section of interest'  
**Default:** 'Inherit from model'

**See also**

- See the Subsystem block reference page for more information.
- For details on how to apply memory sections to atomic subsystems, see “Override Default Memory Placement for Subsystem Functions and Data” (Embedded Coder).

**Memory section for internal data — Select how to apply memory sections**

Select how Embedded Coder applies memory sections to the subsystem internal data.

- **Inherit from model**  
  Apply the root model's memory sections to the subsystem's data

- **Default**  
  Not apply memory sections to the subsystem's data, overriding any model-level specification

- **The memory section of interest**  
  Apply one of the model's memory sections to the subsystem

**Tips**

- The memory section that you specify applies to the corresponding global data structures in the generated code. For basic information about the global data structures generated for atomic subsystems, see “Standard Data Structures in the Generated Code” (Simulink Coder).
- Can be useful for overriding the model's memory section settings for the given subsystem.
- The possible values vary depending on what (if any) package of memory sections you have set for the model's configuration. See “Control Data and Function Placement in Memory by Inserting Pragmas” (Embedded Coder).
- If you have not configured the model with a package, Inherit from model is the only value that appears. Otherwise, the list includes Default and all memory sections the model's package contains.
Dependency

- This parameter requires a license for Embedded Coder and an ERT-based system target file.
- To enable this parameter, set Function packaging to Nonreusable function and select the Function with separate data parameter.

Programmatic Use
Parameter: RTWMemSecDataInternal
Type: character vector
Value: 'Inherit from model' | 'Default' | 'The memory section of interest'
Default: 'Inherit from model'

See also

- See the Subsystem block reference page for more information.
- For details on how to apply memory sections to atomic subsystems, see “Override Default Memory Placement for Subsystem Functions and Data” (Embedded Coder).

Memory section for parameters — Select how to apply memory sections

Inherit from model (default) | Default | The memory section of interest

Select how Embedded Coder applies memory sections to the subsystem parameters.

Inherit from model
- Apply the root model's memory sections to the subsystem's function code

Default
- Not apply memory sections to the subsystem's system code, overriding any model-level specification

The memory section of interest
- Apply one of the model's memory sections to the subsystem

Tips

- The memory section that you specify applies to the corresponding global data structure in the generated code. For basic information about the global data structures generated for atomic subsystems, see “Standard Data Structures in the Generated Code” (Simulink Coder).
- Can be useful for overriding the model's memory section settings for the given subsystem.
• The possible values vary depending on what (if any) package of memory sections you have set for the model's configuration. See “Control Data and Function Placement in Memory by Inserting Pragmas” (Embedded Coder).

• If you have not configured the model with a package, Inherit from model is the only value that appears. Otherwise, the list includes Default and all memory sections the model's package contains.

Dependency

• This parameter requires a license for Embedded Coder and an ERT-based system target file.

• To enable this parameter, set Function packaging to Nonreusable function and select the Function with separate data parameter.

Programmatic Use
Parameter: RTWMemSecDataParameters
Type: character vector
Value: 'Inherit from model' | 'Default' | 'The memory section of interest'
Default: 'Inherit from model'

See also

• See the Subsystem block reference page for more information.

• For details on how to apply memory sections to atomic subsystems, see “Override Default Memory Placement for Subsystem Functions and Data” (Embedded Coder).

Subsystem Reference

Convert to Subsystem Reference — Save a subsystem in a separate file
string | character vector

The Referenced subsystem parameter helps you to convert a Subsystem block to a referenced subsystem.

For more information on how to convert a subsystem to a referenced subsystem, see “Convert an Existing Subsystem to a Referenced Subsystem”.
Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>bus</th>
<th>double</th>
<th>enumerated</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
<th>string</th>
</tr>
</thead>
</table>

Direct Feedthrough          | no      |

Multidimensional Signals    | limited |

Variable-Size Signals       | limited |

Zero-Crossing Detection     | no      |

---

a. Actual data type or capability support depends on block implementation.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Module (default)</td>
<td>Generate code for the subsystem and the blocks within the subsystem.</td>
</tr>
</tbody>
</table>
Architecture | Description
--- | ---
BlackBox | Generate a black box interface. The generated HDL code includes only the input/output port definitions for the subsystem. Therefore, you can use a subsystem in your model to generate an interface to existing, manually written HDL code.

The black-box interface generation for subsystems is similar to the Model block interface generation without the clock signals.

No HDL | Remove the subsystem from the generated code. You can use the subsystem in simulation, however, treat it as a “no-op” in the HDL code.

**Black Box Interface Customization**

For the BlackBox architecture, you can customize port names and set attributes of the external component interface. See “Customize Black Box or HDL Cosimulation Interface” (HDL Coder).

**HDL Block Properties**

<table>
<thead>
<tr>
<th>General</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AdaptivePipelining</td>
<td>Automatic pipeline insertion based on the synthesis tool, target frequency, and multiplier word-lengths. The default is inherit. See also “AdaptivePipelining” (HDL Coder).</td>
</tr>
<tr>
<td>BalanceDelays</td>
<td>Detects introduction of new delays along one path and inserts matching delays on the other paths. The default is inherit. See also “BalanceDelays” (HDL Coder).</td>
</tr>
<tr>
<td>ClockRatePipelining</td>
<td>Insert pipeline registers at a faster clock rate instead of the slower data rate. The default is inherit. See also “ClockRatePipelining” (HDL Coder).</td>
</tr>
<tr>
<td>ConstrainedOutputPipeline</td>
<td>Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. For more details, see “ConstrainedOutputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td>DistributedPipelining</td>
<td>Pipeline register distribution, or register retiming. The default is off. See also “DistributedPipelining” (HDL Coder).</td>
</tr>
<tr>
<td>DSPStyle</td>
<td>Synthesis attributes for multiplier mapping. The default is none. See also “DSPStyle” (HDL Coder).</td>
</tr>
</tbody>
</table>
### General

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FlattenHierarchy</td>
<td>Remove subsystem hierarchy from generated HDL code. The default is inherit. See also “FlattenHierarchy” (HDL Coder).</td>
</tr>
<tr>
<td>InputPipeline</td>
<td>Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “InputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td>OutputPipeline</td>
<td>Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “OutputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td>SharingFactor</td>
<td>Number of functionally equivalent resources to map to a single shared resource. The default is 0. See also “Resource Sharing” (HDL Coder).</td>
</tr>
<tr>
<td>StreamingFactor</td>
<td>Number of parallel data paths, or vectors, that are time multiplexed to transform into serial, scalar data paths. The default is 0, which implements fully parallel data paths. See also “Streaming” (HDL Coder).</td>
</tr>
</tbody>
</table>

If this block is not the DUT, the block property settings in the Target Specification tab are ignored. In the HDL Workflow Advisor, if you use the IP Core Generation workflow, these target specification block property values are saved with the model. If you specify these target specification block property values using hdlset_param, when you open HDL Workflow Advisor, the fields are populated with the corresponding values.

### Target Specification

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ProcessorFPGA Synchronization</td>
<td>Processor/FPGA synchronization mode, specified as a character vector. To save this block property on the model, specify the Processor/FPGA Synchronization in the Set Target Interface task of the IP Core Generation workflow. Values: Free running (default)</td>
</tr>
<tr>
<td>Example: 'Free running'</td>
<td></td>
</tr>
<tr>
<td><strong>Target Specification</strong></td>
<td></td>
</tr>
<tr>
<td>------------------------------------------</td>
<td></td>
</tr>
<tr>
<td><strong>TestPointMapping</strong></td>
<td></td>
</tr>
<tr>
<td>To save this block property on the model, specify the mapping of test point ports to target platform interfaces in the Set Target Interface task of the IP Core Generation workflow.</td>
<td></td>
</tr>
<tr>
<td>Values: '' (default)</td>
<td>cell array of character vectors</td>
</tr>
<tr>
<td>Example: ' {{'TestPoint','AXI4-Lite','x&quot;108&quot;'}}'</td>
<td></td>
</tr>
<tr>
<td><strong>TunableParameter Mapping</strong></td>
<td></td>
</tr>
<tr>
<td>To save this block property on the model, specify the mapping of tunable parameter ports to target platform interfaces in the Set Target Interface task of the IP Core Generation workflow.</td>
<td></td>
</tr>
<tr>
<td>Values: '' (default)</td>
<td>cell array of character vectors</td>
</tr>
<tr>
<td>Example: ' {{'myParam','AXI4-Lite','x&quot;108&quot;'}}'</td>
<td></td>
</tr>
<tr>
<td><strong>AXI4RegisterReadback</strong></td>
<td></td>
</tr>
<tr>
<td>To save this block property on the model, specify whether you want to enable readback on AXI4 slave write registers in the Generate RTL Code and IP Core task of the IP Core Generation workflow. To learn more, see “Model Design for AXI4 Slave Interface Generation“ (HDL Coder).</td>
<td></td>
</tr>
<tr>
<td>Values: 'off' (default)</td>
<td>'on'</td>
</tr>
<tr>
<td><strong>AXI4SlaveIDWidth</strong></td>
<td></td>
</tr>
<tr>
<td>To save this block property on the model, specify the number of AXI Master interfaces that you want to connect the DUT IP core to by using the AXI4 Slave ID Width setting in the Generate RTL Code and IP Core task of the IP Core Generation workflow. To learn more, see “Define Multiple AXI Master Interfaces in Reference Designs to access DUT AXI4 Slave Interface“ (HDL Coder).</td>
<td></td>
</tr>
<tr>
<td>Values: 'off' (default)</td>
<td>'on'</td>
</tr>
<tr>
<td><strong>AXI4SlavePortToPipelineRegisterRatio</strong></td>
<td></td>
</tr>
<tr>
<td>To save this block property on the model, specify the number of AXI4 slave ports for which you want a pipeline register to be inserted by using the AXI4 Slave port to pipeline register ratio setting in the Generate RTL Code and IP Core task of the IP Core Generation workflow. To learn more, see “Model Design for AXI4 Slave Interface Generation“ (HDL Coder).</td>
<td></td>
</tr>
<tr>
<td>Values: 'off' (default)</td>
<td>'on'</td>
</tr>
</tbody>
</table>
### Target Specification

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
</table>
| **GenerateDefaultAXI4Slave** | To save this block property on the model, specify whether you want to disable generation of default AXI4 slave interfaces in the Generate RTL Code and IP Core task of the IP Core Generation workflow.  
Values: 'on' (default) | 'off'                                                                                                                                         |
| **IPCoreAdditionalFiles**  | Verilog or VHDL files for black boxes in your design. Specify the full path to each file, and separate file names with a semicolon (;).  
You can set this property in the HDL Workflow Advisor, in the Additional source files field.  
Values: ' ' (default) | character vector                                                                                                                              |
| **IPCoreName**             | IP core name, specified as a character vector.  
You can set this property in the HDL Workflow Advisor, in the IP core name field. If this property is set to the default value, the HDL Workflow Advisor constructs the IP core name based on the name of the DUT.  
Values: ' ' (default) | character vector                                                                                                                              |
| **IPCoreVersion**          | IP core version number, specified as a character vector.  
You can set this property in the HDL Workflow Advisor, in the IP core version field. If this property is set to the default value, the HDL Workflow Advisor sets the IP core version.  
Values: ' ' (default) | character vector                                                                                                                              |

**Example:**

- 'C:\myprojfiles\led_blinking_file1.vhd;C:\myprojfiles\led_blinking_file2.vhd;'
Restrictions

If your DUT is a masked subsystem, you can generate code only if it is at the top level of the model.

For more information, see:

- “External Component Interfaces” (HDL Coder)
- “Generate Black Box Interface for Subsystem” (HDL Coder)

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

Design and simulate fixed-point systems using Fixed-Point Designer™.

See Also

Enabled Subsystem | Enabled and Triggered Subsystem | Function-Call Subsystem | Triggered Subsystem

Topics

“Create a Subsystem”
“Using Function-Call Subsystems”
“Export-Function Models Overview”

Introduced in R2007a
Add, Subtract, Sum of Elements, Sum

Add or subtract inputs

Library:
- Simulink / Math Operations
- HDL Coder / HDL Floating Point Operations
- HDL Coder / Math Operations

Description

The Sum block performs addition or subtraction on its inputs. The Add, Subtract, Sum of Elements, and Sum blocks are identical blocks. This block can add or subtract scalar, vector, or matrix inputs. It can also collapse the elements of a signal and perform a summation.

You specify the operations of the block with the List of signs parameter with plus (+), minus (-), and spacer (|).

- The number of + and - characters equals the number of inputs. For example, +++ requires three inputs. The block subtracts the second (middle) input from the first (top) input, and then adds the third (bottom) input.
- A spacer character creates extra space between ports on the block icon.
- If performing only addition, you can use a numerical value equal to the number of inputs.
- If only there is only one input port, a single + or - adds or subtracts the elements over all dimensions or in the specified dimension.

The Sum block first converts the input data type to its accumulator data type, then performs the specified operations. The block converts the result to its output data type using the specified rounding and overflow modes.

Calculation of Block Output

Output calculation for the Sum block depends on the number of block inputs and the sign of input ports:
If the Sum block has...

<table>
<thead>
<tr>
<th>And...</th>
<th>The formula for output calculation is...</th>
<th>Where...</th>
</tr>
</thead>
<tbody>
<tr>
<td>One input port</td>
<td>The input port sign is +</td>
<td>( y = e[0] + e[1] + e[2] \ldots + e[m] )</td>
</tr>
<tr>
<td></td>
<td>The input port sign is -</td>
<td>( y = 0.0 - e[0] - e[1] - e[2] \ldots - e[m] )</td>
</tr>
<tr>
<td>Two or more input ports</td>
<td>All input port signs are -</td>
<td>( y = 0.0 - u[0] - u[1] - u[2] \ldots - u[n] )</td>
</tr>
<tr>
<td></td>
<td>The ( k^{th} ) input port is the first port where the sign is +</td>
<td>( y = u[k] - u[0] - u[1] - u[2] - u[k-1] (+/-) u[k+1] \ldots (+/-) u[n] )</td>
</tr>
</tbody>
</table>

### Ports

### Inputs

The inputs can be of different data types, unless you select the **Require all inputs to have the same data type** parameter.

**Port_1 — First input operand signal**

**scalar | vector | matrix**

Input signal to the addition or subtraction operation. If there is only one input signal, then addition or subtraction is performed on the elements over all dimensions or the specified dimension.

Data Types: **single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point**

**Port_n — nth input operand signal**

**scalar | vector | matrix**

nth input signal to the operations. The number of inputs matches the number of signs in the **List of signs** parameter. The block applies the operations to the inputs in the order...
listed. You can also use a numerical value equal to the number of input ports as the **List of signs** parameter. The block creates the input ports and applies addition to all inputs. For example, if you assign 5 for the **List of signs** parameter, the block creates 5 input ports and adds them together to produce the output.

All nonscalar inputs must have the same dimensions. Scalar inputs are expanded to have the same dimensions as other inputs.

**Data Types:** single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point

**Output**

**Port_1 — Output signal**

scalar | vector | matrix

Output signal resulting from addition and/or subtraction operations. The output signal has the same dimension as the input signals.

**Data Types:** single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point

**Parameters**

**Main**

**Icon shape — Block icon shape**

rectangular (default) | round

Designate the icon shape of the block as rectangular or round.

For a rectangular block, the first input port is the top port. For a round Sum block, the first input port is the port closest to the 12 o'clock position going in a counterclockwise direction around the block. Similarly, other input ports appear in counterclockwise order around the block.

**Programmatic Use**

**Block Parameter:** IconShape

**Type:** character vector

**Values:** 'rectangular' | 'round'
**List of signs — Operations performed on inputs**

Enter addition and subtraction operations performed on the inputs. An input port is created for each operation. A spacer (|) creates extra space between the input ports on the block icon. Addition is the default operation. If you only want to add the inputs, enter the number of input ports. The operations are performed in the order listed.

When you enter only one element, the block enables the **Sum over** parameter. For a single vector input, + or - adds or subtracts the elements over all dimensions or in the specified dimension.

**Tips**

You can manipulate the positions of the input ports on the block by inserting spacers (|) between the signs in the **List of signs** parameter. For example, “++|--” creates an extra space between the second and third input ports.

**Programmatic Use**

**Block Parameter:** Inputs

**Type:** character vector

**Values:** '+ | - | | | integer

**Default:** '++'

**Sum over — Dimensions for operations on a single vector input**

**All dimensions** (default) | **Specified dimension**

Select the dimension over which the block performs the sum-over operation.

For **All dimensions**, all input elements are summed. When you select configuration parameter **Use algorithms optimized for row-major array layout**, Simulink enables row-major algorithms for simulation. To generate row-major code, set configuration parameter **Array layout** (Simulink Coder) to **Row-major** in addition to selecting **Use algorithms optimized for row-major array layout**. The column-major and row-major algorithms differ only in the summation order. In some cases, due to different operation order on the same data set, you might experience minor numeric differences in the outputs of column-major and row-major algorithms.

When you select **Specified dimensions**, another parameter **Dimension** appears. Choose the specific dimension for summing the vector input.
Dependency

Enabled when you list only one sign in the List of signs parameter.

Programmatic Use

Block Parameter: CollapseMode
Type: character vector
Values: 'All dimensions' | 'Specified dimension'
Default: 'All dimensions'

Dimension — Dimension for summation on vector input
1 (default) | integer

When you choose Specified dimension for the Sum over parameter, specify the dimension over which to perform the operation.

The block follows the same summation rules as the MATLAB sum function.

Suppose that you have a 2-by-3 matrix $U$.

• Setting Dimension to 1 results in the output $Y$ being computed as:

$$ Y = \sum_{i=1}^{2} U(i, j) $$

• Setting Dimension to 2 results in the output $Y$ being computed as:

$$ Y = \sum_{j=1}^{3} U(i, j) $$

If the specified dimension is greater than the dimension of the input, an error message appears.

Dependency

Enabled when you choose Specified dimension for the Sum over parameter.

Programmatic Use

Block Parameter: CollapseDim
Type: character vector
Value: integer
Default: '1'

Sample time — Specify sample time as a value other than -1
-1 (default) | scalar | vector
Specify the sample time as a value other than -1. For more information, see “Specify Sample Time”.

**Dependencies**

This parameter is not visible unless it is explicitly set to a value other than -1. To learn more, see “Blocks for Which Sample Time Is Not Recommended”.

**Programmatic Use**

**Block Parameter:** SampleTime  
**Type:** character vector  
**Values:** scalar or vector  
**Default:** '-1'

**Signal Attributes**

Click the **Show data type assistant** button to display the **Data Type Assistant**, which helps you set the data type attributes. For more information, see “Specify Data Types Using Data Type Assistant”.

**Require all inputs to have the same data type — Require that all inputs have the same data type**

off (default) | on

Specify if input signals must all have the same data type. If you enable this parameter, then an error occurs during simulation if the input signal types are different.

**Programmatic Use**

**Block Parameter:** InputSameDT  
**Type:** character vector  
**Values:** 'off' | 'on'  
**Default:** 'off'

**Accumulator data type — Data type of the accumulator**

Inherit: Inherit via internal rule (default) | Inherit: Same as first input | double | single | int8 | uint8 | int16 | uint16 | int32 | uint32 | int64 | uint64 | fixdt(1,16) | fixdt(1,16,0) | fixdt(1,16,2^0,0) | <data type expression>

Choose the data type of the accumulator. The type can be inherited, specified directly, or expressed as a data type object such as `Simulink.NumericType`. When you choose
Inherit: Inherit via internal rule, Simulink chooses a data type to balance numerical accuracy, performance, and generated code size, while taking into account the properties of the embedded target hardware.

**Programmatic Use**
**Block Parameter:** AccumDataTypeStr  
**Type:** character vector  
**Values:** 'Inherit: Inherit via internal rule'| 'Inherit: Same as first input' | 'double' | 'single' | 'int8' | 'uint8' | 'int16' | 'uint16' | 'int32' | 'uint32' | 'int64' | 'uint64' | 'fixdt(1,16)' | 'fixdt(1,16,0)' | 'fixdt(1,16,2^0,0)' | '<data type expression>'  
**Default:** 'Inherit: Inherit via internal rule'

**Output minimum — Minimum output value for range checking**
[] (default) | scalar  

Lower value of the output range that Simulink checks.

Simulink uses the minimum to perform:

- Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters”) for some blocks.
- Simulation range checking (see “Signal Ranges” and “Enable Simulation Range Checking”).
- Automatic scaling of fixed-point data types.
- Optimization of the code that you generate from the model. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. For more information, see “Optimize using the specified minimum and maximum values” (Simulink Coder).

**Note** Output minimum does not saturate or clip the actual output signal. Use the Saturation block instead.

**Programmatic Use**
**Block Parameter:** OutMin  
**Type:** character vector  
**Values:** '[]' | scalar  
**Default:** '[]'
**Output maximum — Maximum output value for range checking**

[] (default) | scalar

Upper value of the output range that Simulink checks.

Simulink uses the maximum value to perform:

- Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters”) for some blocks.
- Simulation range checking (see “Signal Ranges” and “Enable Simulation Range Checking”).
- Automatic scaling of fixed-point data types.
- Optimization of the code that you generate from the model. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. For more information, see “Optimize using the specified minimum and maximum values” (Simulink Coder).

**Note** Output maximum does not saturate or clip the actual output signal. Use the Saturation block instead.

**Programmatic Use**

**Block Parameter:** OutMax

**Type:** character vector

**Values:** '[]' | scalar

**Default:** '[]'

**Output data type — Specify the output data type**

Inherit: Inherit via internal rule (default) | Inherit: Keep MSB | Inherit: Keep LSB | Inherit: Inherit via back propagation | Inherit: Same as first input | Inherit: Same as accumulator | double | single | int8 | uint8 | int16 | uint16 | int32 | uint32 | int64 | uint64 | fixdt(1,16) | fixdt(1,16,0) | fixdt(1,16,2^0,0) | <data type expression>

Choose the data type for the output. The type can be inherited, specified directly, or expressed as a data type object such as Simulink.NumericType.

When you select an inherited option, the block behaves as follows:
• **Inherit: Inherit via internal rule**—Simulink chooses a data type to balance numerical accuracy, performance, and generated code size, while taking into account the properties of the embedded target hardware.

**Note** The accumulator internal rule favors greater numerical accuracy, possibly at the cost of less efficient generated code. To get the same accuracy for the output, set the output data type to **Inherit: Inherit same as accumulator**.

• **Inherit: Keep MSB**—Simulink chooses a data type that maintains the full range of the operation, then reduces the precision of the output to a size appropriate for the embedded target hardware.

**Tip** For more efficient generated code, set the **Accumulator data type** to **Inherit: Inherit via internal rule**, and deselect the **Saturate on integer overflow** parameter.

This rule never produces overflows.

• **Inherit: Keep LSB**—Simulink chooses a data type that maintains the precision of the operation, but reduces the range if the full type does not fit on the embedded target hardware.

**Tip** For more efficient generated code, set the **Accumulator data type** to **Inherit: Inherit via internal rule**, and deselect the **Saturate on integer overflow** parameter.

This rule can produce overflows.

If you change the embedded target settings, the data type selected by these internal rules might change. It is not always possible for the software to optimize code efficiency and numerical accuracy at the same time. If the rules do not meet your specific needs for numerical accuracy or performance, use one of the following options:

• Specify the output data type explicitly.
• Use the simple choice of **Inherit: Same as first input**.
• Explicitly specify a default data type such as `fixpt(1,32,16)` and then use the Fixed-Point Tool to propose data types for your model. For more information, see `fxptdlg`.

• To specify your own inheritance rule, use `Inherit: Inherit via back propagation` and then use a Data Type Propagation block. Examples of how to use this block are available in the Signal Attributes library Data Type Propagation Examples block.

• `Inherit: Inherit via back propagation` — Use data type of the driving block.

• `Inherit: Same as first input` — Use data type of the first input signal.

• `Inherit: Inherit same as accumulator` — Use data type of the accumulator.

**Programmatic Use**

**Block Parameter:** OutDataTypeStr
**Type:** character vector

**Values:**
- `'Inherit: Inherit via internal rule'`
- `'Inherit: Keep MSB'`
- `'Inherit: Keep LSB'`
- `'Inherit: Inherit via back propagation'`
- `'Inherit: Same as first input'`
- `'Inherit: Same as accumulator'`
- `'double'`
- `'single'`
- `'int8'`
- `'uint8'`
- `'int16'`
- `'uint16'`
- `'int32'`
- `'uint32'`
- `'int64'`
- `'uint64'`
- `'fixpt(1,16)'`
- `'fixpt(1,16,2^0,0)'`
- `<data type expression>`

**Default:** `'Inherit: Inherit via internal rule'`

**Lock data type settings against changes by the fixed-point tools — Prevent fixed-point tools from overriding data types**

**off** (default) | **on**

Select to lock data type settings of this block against changes by the Fixed-Point Tool and the Fixed-Point Advisor. For more information, see “Lock the Output Data Type Setting” (Fixed-Point Designer).

**Programmatic Use**

**Block Parameter:** LockScale

**Values:**
- `'off'`
- `'on'`

**Default:** `'off'`

**Integer rounding mode — Rounding mode for fixed-point operations**

**Floor** (default) | **Ceiling** | **Convergent** | **Nearest** | **Round** | **Simplest** | **Zero**

Specify the rounding mode for fixed-point operations. For more information, see “Rounding” (Fixed-Point Designer).
Block parameters always round to the nearest representable value. To control the rounding of a block parameter, enter an expression using a MATLAB rounding function into the mask field.

**Programmatic Use**

**Block Parameter:** RndMeth

**Type:** character vector

**Values:** 'Ceiling' | 'Convergent' | 'Floor' | 'Nearest' | 'Round' | 'Simplest' | 'Zero'

**Default:** 'Floor'

**Saturate on integer overflow — Method of overflow action**

off (default) | on

Specify whether overflows saturate or wrap.

<table>
<thead>
<tr>
<th>Action</th>
<th>Rationale</th>
<th>Impact on Overflows</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Select this check box (on).</td>
<td>Your model has possible overflow, and you want explicit saturation protection in the generated code.</td>
<td>Overflows saturate to either the minimum or maximum value that the data type can represent.</td>
<td>The maximum value that the int8 (signed, 8-bit integer) data type can represent is 127. Any block operation result greater than this maximum value causes overflow of the 8-bit integer. With the check box selected, the block output saturates at 127. Similarly, the block output saturates at a minimum output value of -128.</td>
</tr>
<tr>
<td>Action</td>
<td>Rationale</td>
<td>Impact on Overflows</td>
<td>Example</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>---------------------------------------------------------------------------</td>
<td>-------------------------------------------------------------------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Do not select this check box</td>
<td>You want to optimize efficiency of your generated code.</td>
<td>Overflows wrap to the appropriate value that is representable by the data type.</td>
<td>The maximum value that the int8 (signed, 8-bit integer) data type can represent is 127. Any block operation result greater than this maximum value causes overflow of the 8-bit integer. With the check box cleared, the software interprets the overflow-causing value as int8, which can produce an unintended result. For example, a block result of 130 (binary 1000 0010) expressed as int8, is -126.</td>
</tr>
<tr>
<td>(off)</td>
<td>You want to avoid overspecifying how a block handles out-of-range signals.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

When you select this check box, saturation applies to every internal operation on the block, not just the output, or result. Usually, the code generation process can detect when overflow is not possible. In this case, the code generator does not produce saturation code.

**Programmatic Use**

**Block Parameter:** SaturateOnIntegerOverflow  
**Type:** character vector  
**Values:** 'off' | 'on'  
**Default:** 'off'

**Block Characteristics**

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>double</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Direct Feedthrough</strong></td>
<td></td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Multidimensional Signals</strong></td>
<td></td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

### HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

#### HDL Architecture of Sum, Add, and Subtract Blocks

The default Linear architecture generates a chain of N operations (adders) for N inputs.

#### HDL Architecture of Sum of Elements Block

For the Sum of Elements block, HDL Coder supports Tree and Cascade architectures for Sum of Elements blocks that have a single vector input with multiple elements.

This block has multi-cycle implementations that introduce additional latency in the generated code. To see the added latency, view the generated model or validation model. See “Generated Model and Validation Model” (HDL Coder).

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Additional cycles of latency</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear</td>
<td>0</td>
<td>Generates a linear chain of adders to compute the sum of products.</td>
</tr>
<tr>
<td>Architecture</td>
<td>Additional cycles of latency</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>-----------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>Tree</td>
<td>0</td>
<td>Generates a tree structure of adders to compute the sum of products.</td>
</tr>
<tr>
<td>Cascade</td>
<td>1, when block has a single vector input port.</td>
<td>This implementation optimizes latency * area and is faster than the Tree implementation. It computes partial sums and cascades adders. See “Cascade Architecture Best Practices” (HDL Coder).</td>
</tr>
</tbody>
</table>

**HDL Block Properties**

**Note** To use the **LatencyStrategy** setting in the **Native Floating Point** tab of the HDL Block Properties dialog box, specify **Linear** or **Tree** as the HDL Architecture.

**General**

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ConstrainedOutputPipeline</strong></td>
<td>Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. For more details, see “ConstrainedOutputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td><strong>InputPipeline</strong></td>
<td>Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “InputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td><strong>OutputPipeline</strong></td>
<td>Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “OutputPipeline” (HDL Coder).</td>
</tr>
</tbody>
</table>

**Note** The Sum of Elements block does not support HDL code generation with double data types in the Native Floating Point mode.

1-2071
### Native Floating Point

<table>
<thead>
<tr>
<th>LatencyStrategy</th>
<th>Specify whether to map the blocks in your design to inherit, Max, Min, Zero, or Custom for the floating-point operator. The default is inherit. See also “LatencyStrategy” (HDL Coder).</th>
</tr>
</thead>
<tbody>
<tr>
<td>NFPCustomLatency</td>
<td>To specify a value, set LatencyStrategy to Custom. HDL Coder adds latency equal to the value that you specify for the NFPCustomLatency setting. See also “NFPCustomLatency” (HDL Coder).</td>
</tr>
</tbody>
</table>

### Complex Data Support

The default Linear implementation supports complex data.

The Tree implementation supports complex data with + for the List of signs block parameter. With native floating point support, the Tree implementation supports complex data with both + and - for List of signs.

### PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

### Fixed-Point Conversion

Design and simulate fixed-point systems using Fixed-Point Designer™.

### See Also

Bias | Divide | Gain

### Topics

“Control Signal Data Types”

### Introduced before R2006a
Switch

Combine multiple signals into single signal

Library:
- Simulink / Commonly Used Blocks
- Simulink / Signal Routing
- HDL Coder / Commonly Used Blocks
- HDL Coder / Signal Routing

Description

The Switch block passes through the first input or the third input signal based on the value of the second input. The first and third inputs are data input. The second input is a control input. Specify the condition under which the block passes the first input by using the Criteria for passing first input and Threshold parameters.

Bus Support

The Switch block is a bus-capable block. The data inputs can be virtual or nonvirtual bus signals subject to the following restrictions:

- All the buses must be equivalent (same hierarchy with identical names and attributes for all elements).
- All signals in a nonvirtual bus input to a Switch block must have the same sample time. The requirement holds even if the elements of the associated bus object specify inherited sample times.

You can use a Rate Transition block to change the sample time of an individual signal, or of all signals in a bus. See “Specify Bus Signal Sample Times” and Bus-Capable Blocks for more information.

You can use an array of buses as an input signal to a Switch block. For details about defining and using an array of buses, see “Combine Buses into an Array of Buses”. When using an array of buses, set the Threshold parameter to a scalar value.
Limitations

- If the data inputs to the Switch block are buses, the element names of both buses must be the same. Using the same element names ensures that the output bus has the same element names no matter which input bus the block selects. To ensure that your model meets this requirement, use a bus object to define the buses and set the **Element name mismatch** diagnostic to **error**. See “Connectivity Diagnostics Overview” for more information.

Ports

Input

**Port_1 — First data input signal**

scalar | vector

First of two data inputs. The block propagates either the first or second data input to the output. The block selects which input to pass based on the control input. Specify the condition for the control input to pass the first input using the **Criteria for passing first input** and **Threshold** parameters.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus

**Port_2 — Control input signal**

scalar | vector

Control signal the block uses to determine whether to pass the first or second data input to the output. If the control input meets the condition set in the **Criteria for passing first input** parameter, then the block passes the first data input. Otherwise, the block passes the second data input.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus

**Port_3 — Second data input signal**

scalar | vector

Second of two data inputs. The block propagates either the first or second data input to the output. The block selects which input to pass based on the control input. Specify the
condition for the control input to pass the first or second input using the **Criteria for passing first input** and **Threshold** parameters.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus

**Output**

**Port_1 — Output signal**

scalar | vector

Output signal propagated from either the first or second input signal, based on the control signal value.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus

**Parameters**

**Main**

**Criteria for passing first input — Selection criteria to pass first data input**

u2 >= Threshold (default) | u2 > Threshold | u2 ~= 0

Select the condition under which the block passes the first data input. If the control input meets the condition set in the **Criteria for passing first input** parameter, the block passes the first input. Otherwise, the block passes the second data input signal from input Port_3.

u2 >= Threshold

Checks whether the control input is greater than or equal to the threshold value.

u2 > Threshold

Checks whether the control input is greater than the threshold value.

u2 ~=0

Checks whether the control input is nonzero.

**Note** The Switch block does not support u2 ~=0 mode for enumerated data types.
Tip

When the control input is a Boolean signal, use one of these combinations of condition and threshold value:

• \( u_2 \geq \text{Threshold} \), where the threshold value equals 1
• \( u_2 > \text{Threshold} \), where the threshold value equals 0
• \( u_2 \neq 0 \)

Otherwise, the Switch block ignores threshold values and uses the Boolean value for signal routing. For a value of 1, the block passes the first input, and for a value of 0, the block passes the third input. A warning message that describes this behavior also appears in the MATLAB Command Window.

Programmatic Use

Block Parameter: Criteria
Type: character vector
Value: 'u2 >= Threshold' | 'u2 > Threshold' | 'u2 ~=0'
Default: 'u2 >= Threshold'

Threshold — Threshold used in criteria
\( \theta \) (default) | scalar

Assign the threshold used in the Criteria for passing first input that determines which input the block passes to the output. \( \text{Threshold} \) must be greater than Output minimum and less than Output maximum.

To specify a nonscalar threshold, use brackets. For example, the following entries are valid:

• \([1\ 4\ 8\ 12]\)
• \([\text{MyColors.Red}, \text{MyColors.Blue}]\)

Dependencies

Setting Criteria for passing first input to \( u_2 \neq 0 \) disables this parameter.

Programmatic Use

Block Parameter: Threshold
Type: character vector
Value: 'off' | 'on'
Default: 'off'
Enable zero-crossing detection — Enable zero-crossing detection
on (default) | Boolean

Select to enable zero-crossing detection. For more information, see “Zero-Crossing Detection” in the Simulink documentation.

Programmatic Use
Block Parameter: ZeroCross
Type: character vector, string
Values: 'off' | 'on'
Default: 'on'

Signal Attributes

Click the Show data type assistant button to display the Data Type Assistant, which helps you set the data type attributes. For more information, see “Specify Data Types Using Data Type Assistant”.

Require all data port inputs to have the same data type — Require data ports to have the same data type
off (default) | on

Require all data inputs to have the same data type.

Programmatic Use
Block Parameter: InputSameDT
Type: character vector
Value: 
Default: '0'

Output minimum — Minimum output value for range checking
[] (default) | scalar

Lower value of the output range that Simulink checks.

Simulink uses the minimum to perform:

- Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters”) for some blocks.
- Simulation range checking (see “Signal Ranges” and “Enable Simulation Range Checking”).
• Automatic scaling of fixed-point data types.
• Optimization of the code that you generate from the model. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. For more information, see “Optimize using the specified minimum and maximum values” (Simulink Coder).

**Note** Output minimum does not saturate or clip the actual output signal. Use the Saturation block instead.

**Programmatic Use**
**Block Parameter:** OutMin  
**Type:** character vector  
**Values:** ' [] ' | scalar  
**Default:** ' [] '

**Output maximum — Maximum output value for range checking**

[] (default) | scalar

Upper value of the output range that Simulink checks.

Simulink uses the maximum value to perform:

• Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters”) for some blocks.
• Simulation range checking (see “Signal Ranges” and “Enable Simulation Range Checking”).
• Automatic scaling of fixed-point data types.
• Optimization of the code that you generate from the model. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. For more information, see “Optimize using the specified minimum and maximum values” (Simulink Coder).

**Note** Output maximum does not saturate or clip the actual output signal. Use the Saturation block instead.
**Programmatic Use**

**Block Parameter**: OutMax  
**Type**: character vector  
**Values**: '\[ \]' | scalar  
**Default**: '\[ \]'  

**Output data type** — **Output data type**  
Inherit: Inherit via internal rule (default) | Inherit: Inherit via back propagation | double | single | int8 | uint8 | int16 | uint16 | int32 | uint32 | int64 | uint64 | fixdt(1,16) | fixdt(1,16,0) | fixdt(1,16,2^0,0) | Enum:  
<class name> | <data type expression>  
Specify the output data type.  
Inherit: Inherit via internal rule  
Uses the following rules to determine the output data type.

<table>
<thead>
<tr>
<th>Data Type of First Input Port</th>
<th>Output Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Has a larger positive range than the third input port</td>
<td>Inherited from the first input port</td>
</tr>
<tr>
<td>Has the same positive range as the third input port</td>
<td>Inherited from the third input port</td>
</tr>
<tr>
<td>Has a smaller positive range than the third input port</td>
<td></td>
</tr>
</tbody>
</table>

Inherit: Inherit via back propagation  
Uses data type of the driving block.  

double  
Specifies output data type is double.  
single  
Specifies output data type is single.  
int8  
Specifies output data type is int8.  
uint8  
Specifies output data type is uint8.
int16
Specifies output data type is int16.

uint16
Specifies output data type is uint16.

int32
Specifies output data type is int32.

uint32
Specifies output data type is uint32.

int64
Specifies output data type is int64.

uint64
Specifies output data type is uint64.

fixdt(1,16,0)
Specifies output data type is fixed point fixdt(1,16,0).

fixdt(1,16,2^0,0)
Specifies output data type is fixed point fixdt(1,16,2^0,0).

Enum: <class name>
Uses an enumerated data type, for example, Enum: BasicColors.

<data type expression>
Uses a data type object, for example, Simulink.NumericType.

Tip
When the output is of enumerated type, both data inputs should use the same enumerated type as the output.

Programmatic Use
Block Parameter: OutDataTypeStr
Type: character vector
Values: 'Inherit: Inherit via internal rule'|'Inherit: Inherit via back propagation'|'double'|'single'|'int8'|'uint8'|'int16'|'uint16'|'int32'|'uint32'|'int64'|'uint64'|'fixdt(1,16)'|'fixdt(1,16,0)'|'fixdt(1,16,2^0,0)'|Enum: <class name> | '<data type expression>'
**Default:** 'Inherit: Inherit via internal rule'

Lock data type settings against changes by the fixed-point tools — Prevent fixed-point tools from overriding data types

off (default) | on

Select this parameter to prevent the fixed-point tools from overriding the data types you specify on this block. For more information, see “Lock the Output Data Type Setting” (Fixed-Point Designer).

**Programmatic Use**

**Block Parameter:** LockScale  
**Type:** character vector  
**Values:** 'off' | 'on'  
**Default:** 'off'

**Integer rounding mode — Specify the rounding mode for fixed-point operations**

Floor (default) | Ceiling | Convergent | Nearest | Round | Simplest | Zero

Choose one of these rounding modes.

**Ceiling**

Rounds both positive and negative numbers toward positive infinity. Equivalent to the MATLAB ceil function.

**Convergent**

Rounds number to the nearest representable value. If a tie occurs, rounds to the nearest even integer. Equivalent to the Fixed-Point Designer convergent function.

**Floor**

Rounds both positive and negative numbers toward negative infinity. Equivalent to the MATLAB floor function.

**Nearest**

Rounds number to the nearest representable value. If a tie occurs, rounds toward positive infinity. Equivalent to the Fixed-Point Designer nearest function.

**Round**

Rounds number to the nearest representable value. If a tie occurs, rounds positive numbers toward positive infinity and rounds negative numbers toward negative infinity. Equivalent to the Fixed-Point Designer round function.
Simplest
Automatically chooses between round toward floor and round toward zero to generate rounding code that is as efficient as possible.

Zero
Rounds number toward zero. Equivalent to the MATLAB fix function.

Programmatic Use
Block Parameter: RndMeth
Type: character vector
Values: 'Ceiling' | 'Convergent' | 'Floor' | 'Nearest' | 'Round' | 'Simplest' | 'Zero'
Default: 'Floor'

See Also
For more information, see “Rounding” (Fixed-Point Designer).

Saturate on integer overflow — Method of overflow action
off (default) | on

Specify whether overflows saturate or wrap.

• off — Overflows wrap to the appropriate value that the data type can represent.

    For example, the number 130 does not fit in a signed 8-bit integer and wraps to -126.

• on — Overflows saturate to either the minimum or maximum value that the data type can represent.

    For example, an overflow associated with a signed 8-bit integer can saturate to -128 or 127.

Tip
• Consider selecting this check box when your model has a possible overflow and you want explicit saturation protection in the generated code.

• Consider clearing this check box when you want to optimize efficiency of your generated code.

    Clearing this check box also helps you to avoid overspecifying how a block handles out-of-range signals. For more information, see “Check for Signal Range Errors”.

1-2082
- When you select this check box, saturation applies to every internal operation on the block, not just the output or result.
- In general, the code generation process can detect when overflow is not possible. In this case, the code generator does not produce saturation code.

**Programmatic Use**  
**Block Parameter:** SaturateOnIntegerOverflow  
**Type:** character vector  
**Values:** 'off' | 'on'  
**Default:** 'off'

Allow different data input sizes — Allow different data input sizes

off (default) | on

Select this check box to allow input signals with different sizes. The block propagates the input signal size to the output signal. If the two data inputs are variable-size signals, the maximum size of the signals can be equal or different.

**Programmatic Use**  
**Block Parameter:** AllowDiffInputSizes  
**Type:** character vector  
**Value:** 'on' | 'off'  
**Default:** 'off'

**Block Characteristics**

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>bus</th>
<th>double</th>
<th>enumerated</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
<th>string</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1-2083
Zero-Crossing Detection | yes

## Extended Capabilities

### C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.
Generated code relies on `memcpy` or `memset` functions (string.h) under certain conditions.

### HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.
HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

### HDL Architecture
This block has a single, default HDL architecture.

### HDL Block Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ConstrainedOutputPipeline</strong></td>
<td>Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. For more details, see “ConstrainedOutputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td><strong>InputPipeline</strong></td>
<td>Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “InputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td><strong>OutputPipeline</strong></td>
<td>Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “OutputPipeline” (HDL Coder).</td>
</tr>
</tbody>
</table>
Complex Data Support
This block supports code generation for complex signals.

PLC Code Generation
Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

See Also
Manual Switch | Multiport Switch

Introduced before R2006a
Switch Case

Select subsystem execution using logic similar to a switch statement.

**Library:** Simulink / Ports & Subsystems

**Description**

The Switch Case block with Switch Case Action Subsystem blocks containing Action Port blocks, implements switch logic to control subsystem execution.

A Switch Case block has a single input. To select a case, define the input value using the **Case conditions** parameter. The cases are evaluated top down starting with the first case.

Each case is associated with an output port that is attached to a Switch Case Action Subsystem block. When a case is selected, the associated output port sends an action signal to execute the subsystem.
A default case is selected after all of the other case conditions evaluate to false. Providing a default case is optional, even if the other case conditions do not exhaust every possible input value.

Cases for the Switch Case block contain an implied break after a Switch Case Action Subsystem block is executed. Therefore, there is no fall through behavior for the Simulink Switch Case block as found in standard C switch statements.

Ports

Input

u1 (logical operator) — Value for case selection
scalar

Input to the port labeled u1 of a Switch Case block can be:

- A scalar value with a built-in data type that Simulink supports. However, the Switch Case block does not support Boolean or fixed-point data types, and it truncates numeric inputs to 32-bit signed integers.
- A scalar value of any enumerated data type.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | enumerated

Output

case — Action signal for a Switch Case Action Subsystem block
scalar

Output from the Case and default ports are action signals connected to Switch Case Action Subsystem blocks.

Parameters

Case conditions — Specify case values
{1} (default) | list of cases
Specify the cases values using MATLAB cell notation.

{1}

Specify the output port labeled case[1] outputs an action signal when the input port value is 1.

List of ports with case assignments

Specify multiple cases and ports using MATLAB cell notation. For example, entering {1, [7, 9, 4]} specifies that output port case[1] is run when the input value is 1, and output port case [7 9 4] is run when the input value is 7, 9, or 4.

You can use colon notation to specify a range of integer case conditions. For example, entering {[1:5]} specifies that output port case[1 2 3 4 5] is run when the input value is 1, 2, 3, 4, or 5.

Depending on block size, cases from a long list of case conditions are displayed in shortened form on the face of the Switch Case block, using a terminating ellipsis (...).

You can use the enumeration function to specify case conditions that include a case for every value in an enumerated type.

**Programmatic Use**

**Block Parameter:** CaseConditions  
**Type:** character vector  
**Values:** '{1}' | '<list of cases>'  
**Default:** '{1}'

**Show default case — Control display of default output port**

*off (default) | on*

Control display of default output port.

*off*

Hide default output port.

*on*

Display default output port as the last case on the Switch Case block. This allows you to specify a default case that executes when the input value does not match any of the other case values.

**Programmatic Use**

**Block Parameter:** ShowDefaultCase
Type: character vector
Values: 'off' | 'on'
Default: 'off'

Enable zero-crossing detection — Control zero-crossing detection
on (default) | off

Control zero-crossing detection.

✅ on
   Detect zero crossings.

☐ off
   Do not detect zero crossings.

Programmatic Use
Block Parameter: ZeroCross
Type: character vector
Values: 'on' | 'off'
Default: 'on'

Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>double</th>
<th>enumerated</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also

Blocks
Action Port | Subsystem | Switch Case Action Subsystem

Topics
Select Subsystem Execution

Introduced before R2006a
Switch Case Action Subsystem

Subsystem whose execution is enabled by a Switch Case block

Library: Simulink / Ports & Subsystems

Description

The Switch Case Action Subsystem block is a Subsystem block preconfigured as a starting point for creating a subsystem whose execution is controlled by a Switch Case block. The input port to a Switch Case block selects a case defined using the **Case conditions** parameter. Depending on input value and case selected, an action signal is sent to execute a Switch Case Action Subsystem block.

All blocks in a Switch Case Action Subsystem block must run at the same rate as the driving Switch Case block. You can achieve this requirement by setting each block sample time parameter to be either inherited (-1) or the same value as the Switch Case block sample time.

**Merge signals from Switch Case Subsystem blocks**

This example shows how to create a one signal from multiple subsystem output signals. The Switch Case block selects the execution of one If Action Subsystem block from a set of subsystems. Regardless of which subsystem the Switch Case block selects, you can create a one resulting signal with a Merge block.
**Ports**

**Input**

**In1 — Signal input to a subsystem block**

scalar | vector | matrix

Placing an Inport block in a subsystem block adds an external input port to the block. The port label matches the name of the Inport block.

Use Inport blocks to get signals from the local environment.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus

**Action — Control signal input to a subsystem block**

scalar | vector | matrix

Placing an Action Port block in a subsystem block adds an external input port to the block and changes the block to a Switch Case Action Subsystem block.

Dot-dash lines from a Switch Case block to an Switch Case Action Subsystem block represent action signals. An action signal is a control signal connected to the action port of an Switch Case Action Subsystem block. A message on the action signal initiates execution of the subsystem.

Data Types: action
Output

Out1 — Signal output from a subsystem
scalar | vector | matrix

Placing an Outport block in a subsystem block adds an output port from the block. The port label on the subsystem block is the name of the Outport block.

Use Outport blocks to send signals to the local environment.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus

Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>bus</th>
<th>double</th>
<th>enumerated</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
<th>string</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>limited</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>limited</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

a. Actual data type or capability support depends on block implementation.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

Actual data type or capability support depends on block implementation.
Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

Actual data type support depends on block implementation.

See Also

Blocks
Action Port | Subsystem | Switch Case

Topics
Select Subsystem Execution

Introduced before R2006a
Tapped Delay

Delay scalar signal multiple sample periods and output all delayed versions

Library: Simulink / Discrete
HDL Coder / Discrete

Description

The Tapped Delay block delays an input by the specified number of sample periods and provides an output signal for each delay. For example, when you specify 4 for **Number of delays** and **Order output starting with** is **Oldest**, the block provides four outputs — the first delayed by four sample periods, the second delayed by three, and so on. Use this block to discretize a signal in time or resample a signal at a different rate.

The block accepts one scalar input and generates an output vector that contains data for each incremental delay. Specify the order of the delayed signals in the output vector with the **Order output vector starting with** parameter:

- **Oldest** orders the output vector starting with the oldest delay version and ending with the newest delay version.
- **Newest** orders the output vector starting with the newest delay version and ending with the oldest delay version.

Specify the output vector for the first sampling period with the **Initial condition** parameter. Careful selection of this parameter can mitigate unwanted output behavior.

Specify the time between samples with the **Sample time** parameter. Specify the number of delays with the **Number of delays** parameter. A value of -1 instructs the block to inherit the number of delays by backpropagation. Each delay is equivalent to the $z^{-1}$ discrete-time operator, which the Unit Delay block represents.
Ports

Input
Port_1 — Input signal
scalar

Input signal to delay.
Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point

Output
Port_1 — Delayed versions of input signal
scalar | vector

All versions of the delayed input signal. Use the Order output vector starting with parameter to specify the order of delayed signals in the output vector.
Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point

Parameters

Initial condition — Initial output
0.0 (default) | scalar

Specify the initial output of the simulation. The Initial condition parameter is converted from a double to the input data type offline using round-to-nearest and saturation.

Limitations

The initial condition of this block cannot be inf or NaN.

Programmatic Use
Block Parameter: vinit
Type: character vector
Values: scalar
Default: '0.0'
Sample time — Time between samples
-1 (default) | scalar | vector

Specify the time interval between samples. To inherit the sample time, set this parameter to -1. For more information, see “Specify Sample Time”.

Programmatic Use
Block Parameter: samptime
Type: character vector
Values: scalar | vector
Default: '-1'

Number of delays — Number of discrete-time operators
4 (default) | positive scalar | -1 (for inherited)

Specify the number of discrete-time operators as a positive scalar, or -1 for inherited.

Programmatic Use
Block Parameter: NumDelays
Type: character vector
Values: positive scalar | -1 (inherited)
Default: '4'

Order output vector starting with — Order of output
Oldest (default) | Newest

Specify whether to output the oldest delay version first, or the newest delay version first.

Programmatic Use
Block Parameter: DelayOrder
Type: character vector
Values: 'Oldest' | 'Newest'
Default: 'Oldest'

Include current input in output vector — Include current input in output vector
off (default) | on

Select this check box to include the current input in the output vector.

Programmatic Use
Block Parameter: includeCurrent
Type: character vector
Values: 'off' | 'on'
Default: 'off'

Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>double</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture
This block has a single, default HDL architecture.
**HDL Block Properties**

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConstrainedOutputPipeline</td>
<td>Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. For more details, see “ConstrainedOutputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td>InputPipeline</td>
<td>Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “InputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td>OutputPipeline</td>
<td>Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “OutputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td>ResetType</td>
<td>Suppress reset logic generation. The default is default, which generates reset logic. See also “ResetType” (HDL Coder).</td>
</tr>
</tbody>
</table>

**Complex Data Support**

This block supports code generation for complex signals.

**PLC Code Generation**

Generate Structured Text code using Simulink® PLC Coder™.

**Fixed-Point Conversion**

Design and simulate fixed-point systems using Fixed-Point Designer™.

**See Also**

Delay | Resettable Delay | Unit Delay | Variable Integer Delay

**Topics**

“Specify Sample Time”
Introduced before R2006a
**Terminate Function**

Execute contents on a model terminate event  
**Library:** Simulink / User-Defined Functions

**Description**

The Terminate Function block is a pre-configured subsystem block that executes on a model terminate event. By default, the Terminate Function block includes an Event Listener block with **Event** set to **Terminate**, a Terminator block, and a State Reader block.

Replace the Terminator block with blocks to save the state value from the State Reader block.

Conditionally executed subsystem blocks with output ports fully supported within Terminate Function blocks

For a list of unsupported blocks and features, see “Initialize, Reset, and Terminate Function Limitations”.

The input and output ports of a component containing Initialize Function and Terminate Function blocks must connect to input and output port blocks.
The code generated from this block is part of the `model_terminate` function that is called once at the end of model execution.

## Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>bus</th>
<th>double</th>
<th>enumerated</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
<th>string</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>limited</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

a. Actual data type or capability support depends on block implementation.

## Extended Capabilities

### C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

### Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

### See Also
Event Listener | Initialize Function | Reset Function | State Reader | State Writer

### Topics
“Using Initialize, Reset, and Terminate Functions”
“Create Test Harness to Generate Function Calls”
“Generate Code That Responds to Initialize, Reset, and Terminate Events” (Simulink Coder)
Terminator

Terminate unconnected output port

**Library:**
- Simulink / Commonly Used Blocks
- Simulink / Sinks
- HDL Coder / Sinks

**Description**

Use the Terminator block to cap blocks whose output ports do not connect to other blocks. If you run a simulation with blocks having unconnected output ports, Simulink issues warning messages. Using Terminator blocks to cap those blocks helps prevent warning messages.

**Ports**

**Input**

**Port_1 — Input signal**

- scalar | vector | matrix | n-D array | bus

Use this port to direct signals from output ports that are otherwise unconnected during a simulation. The port accepts real or complex signals of all data types.

**Data Types:** single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus

**Block Characteristics**

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>bus</th>
<th>double</th>
<th>enumerated</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
<th>string</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Extended Capabilities

#### C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

#### HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

This block can be used to terminate simulation when used with subsystems that generate HDL code, but is not included in the hardware implementation.

#### PLC Code Generation
Generate Structured Text code using Simulink® PLC Coder™.

#### Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

#### See Also
Ground | “Unconnected block output ports”

#### Topics
“Model Configuration Parameters: Connectivity Diagnostics”

---

<table>
<thead>
<tr>
<th>Multidimensional Signals</th>
<th>yes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
</tr>
</tbody>
</table>
“Systematic Diagnosis of Errors and Warnings”

Introduced before R2006a
Timed-Based Linearization

Generate linear models in base workspace at specific times

Library: Simulink / Model-Wide Utilities

Description

This block calls linmod or dlinmod to create a linear model for the system when the simulation clock reaches the time specified by the Linearization time parameter. No trimming is performed. The linear model is stored in the base workspace as a structure, along with information about the operating point at which the snapshot was taken. Multiple snapshots are appended to form an array of structures.

The block sets the following model parameters to the indicated values:

- BufferReuse = 'off'
- RTWInlineParameters = 'on'
- BlockReductionOpt = 'off'

The name of the structure used to save the snapshots is the name of the model appended by _Timed_Based Linearization, for example, vdp_Timed_Based Linearization. The structure has the following fields:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>The A matrix of the linearization</td>
</tr>
<tr>
<td>b</td>
<td>The B matrix of the linearization</td>
</tr>
<tr>
<td>c</td>
<td>The C matrix of the linearization</td>
</tr>
<tr>
<td>d</td>
<td>The D matrix of the linearization</td>
</tr>
<tr>
<td>StateName</td>
<td>Names of the model's states</td>
</tr>
<tr>
<td>OutputName</td>
<td>Names of the model's output ports</td>
</tr>
<tr>
<td>Field</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>InputName</td>
<td>Names of the model's input ports</td>
</tr>
<tr>
<td>OperPoint</td>
<td>A structure that specifies the operating point of the linearization. The structure specifies the operating point time (OperPoint.t). The states (OperPoint.x) and inputs (OperPoint.u) fields are not used.</td>
</tr>
<tr>
<td>Ts</td>
<td>The sample time of the linearization for a discrete linearization</td>
</tr>
</tbody>
</table>

**Tip** To generate models conditionally, use the Trigger-Based Linearization block.

**Parameters**

**Linearization time — Time at which to generate a linear model**

1 (default) | scalar | vector

Time at which you want the block to generate a linear model. Enter a vector of times if you want the block to generate linear models at more than one time step.

Programmatic Use

Block Parameter: LinearizationTime

Type: character vector

Values: scalar | vector

Default: '1'

**Sample time (of linearized model) — Sample time**

0 (default) | scalar | vector

Specify a sample time for the linear model. To create a continuous-time model using linmod, specify a sample time of 0. Otherwise, to create a discrete-time model using dlinmod, specify a positive sample time (see “Discrete-Time System Linearization” on page 2-58).

Programmatic Use

Block Parameter: SampleTime

Type: character vector

Values: scalar | vector

Default: '0'
Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>no</td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
</tr>
</tbody>
</table>

See Also
Trigger-Based Linearization | dlinmod | linmod

Topics
“Discrete-Time System Linearization” on page 2-58

Introduced in R2010a
To File

Write data to file
Library: Simulink / Sinks
        HDL Coder / Sinks

Description

The To File block writes input signal data into a MAT-file. The block writes to the output file incrementally, with minimal memory overhead during simulation. If the output file exists when the simulation starts, the block overwrites the file. The file automatically closes when you pause the simulation or the simulation completes. If simulation terminates abnormally, the To File block saves the data it has logged up until the point of the abnormal termination.

The To File block icon shows the name of the output file.

Control Amount of Data Saved

If you specify data logging intervals with the Configuration Parameters > Data Import/Export > Logging intervals parameter, the To File block logs only data inside of the intervals. For example, the block logs no data if the intervals are empty ([ ]). The block stores the logged data in the file associated with the block instead of in the variable that you specify for the Single simulation output parameter.

For variable-step solvers, to control the amount of data available to the To File block, use the Configuration Parameters > Data Import/Export > Additional parameters > Output options parameter. For example, to write data at identical time points over multiple simulations, select the Produce specified output only option.

Block parameters also control the amount of data saved. See “Decimation” on page 1-0 and “Sample time” on page 1-0.

Pause Simulation

After pausing a simulation, do not alter any file that a To File block logs into. For example, do not save such a file with the MATLAB save command. Altering the file can cause an
error when you resume the simulation. If you want to alter the file after pausing, copy the file and work with the copy of the file.

If you pause using the Simulation Stepper, the To File block captures the simulation data up to the point of the pause. When you step back, the To File data file no longer contains any simulation data past the new reduced time of the last output.

**Limitations**

When a To File block is in a referenced model, that model must be a single-instance model. Only one instance of such a model can exist in a model hierarchy. See “Model Reuse” for more information.

**Ports**

**Input**

**Port 1 — Input signal**

Scalar | Vector | Matrix | N-D Array

Signal to store in file. Each sample consists of a timestamp and an associated data value. The data can be in array format or MATLAB timeseries format. The To File block accepts real or complex signal data of any data type that Simulink software supports, except fixed-point data with a word length greater than 32 bits.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus

**Parameters**

**File name — Path or file name**

Untitled.mat (default) | MAT-file path or name

Specify the path or file name of the MAT-file in which to store the output. On UNIX systems, the path name can start with a tilde (~) character signifying your home folder. If you specify a file name without path information, Simulink software stores the file in the
MATLAB working folder. (To determine the working folder, at the MATLAB command line, enter `pwd`.) If the file exists, Simulink software overwrites it.

**Programmatic Use**

**Block Parameter:** FileName  
**Type:** character vector  
**Values:** MAT-file path or name  
**Default:** 'untitled.mat'

**Variable name — Matrix name**

*ans (default)* | character vector

Specify the name of the matrix contained in the file.

**Programmatic Use**

**Block Parameter:** MatrixName  
**Type:** character vector  
**Values:** character vector  
**Default:** 'ans'

**Save format — Data format**

*Timeseries (default)* | Array

Specify the data format that the To File block uses for writing data.

Use the *Array* format only for vector, double, noncomplex signals.

For the *Timeseries* format, the To File block:

- Writes data in a MATLAB timeseries object.
- Supports writing multidimensional, real, or complex output values.
- Supports writing output values that have any built-in data type, including Boolean, enumerated (enum), and fixed-point data with a word length of up to 32 bits.
- For virtual and nonvirtual bus input signals, creates a MATLAB structure that matches the bus hierarchy. Each leaf of the structure is a MATLAB timeseries object.

For the *Array* format, the To File block:

- Writes data into a matrix containing two or more rows. The matrix has the following form:
Simulink software writes one column to the matrix for each data sample. The first element of the column contains the timestamp. The remainder of the column contains data for the corresponding output values.

- Supports writing data that is one-dimensional, double, and noncomplex.

The From File block can use data written by a To File block in any format (*Timeseries* or *Array*) without any modifications to the data or other special provisions.

The From Workspace block can read data that is in the *Array* format and is the transposition of the data written by the To File block. To provide the required format, use MATLAB commands to load and transpose the data from the MAT-file.

The following table shows how simulation mode support depends on the **Save format** value.

<table>
<thead>
<tr>
<th>Simulation Mode</th>
<th>Timeseries</th>
<th>Array</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal</td>
<td>Supported</td>
<td>Supported</td>
</tr>
<tr>
<td>Accelerator</td>
<td>Supported</td>
<td>Supported</td>
</tr>
<tr>
<td>Rapid Accelerator</td>
<td>Supported</td>
<td>Supported</td>
</tr>
<tr>
<td>Software-in-the-loop (SIL)</td>
<td>Not supported</td>
<td>Supported if MAT-file logging is enabled</td>
</tr>
<tr>
<td>Processor-in-the-loop (PIL)</td>
<td>Not supported</td>
<td>Supported if MAT-file logging is available and enabled</td>
</tr>
<tr>
<td>External</td>
<td>Not supported</td>
<td>Supported if MAT-file logging is enabled</td>
</tr>
<tr>
<td>RSim target</td>
<td>Supported</td>
<td>Supported if MAT-file logging is enabled</td>
</tr>
</tbody>
</table>

**Programmatic Use**

**Block Parameter:** SaveFormat  
**Type:** character vector
**Decimation** — Decimation factor that determines when data writes
1 (default) | scalar | vector

Specify the decimation factor, \( n \), that writes data at every \( n \)th time that the block executes. The default value has this block writing data at every time step.

**Programmatic Use**
**Block Parameter:** Decimation
**Type:** character vector
**Values:** scalar | vector
**Default:** '1'

**Sample time** — Sample period and offset
-1 (default) | scalar | vector

Specifies the sample period and offset at which to collect data points. This parameter is useful when you are using a variable-step solver where the interval between time steps is not constant. The default value causes the block to inherit the sample time from the driving block. See “Specify Sample Time”.

**Programmatic Use**
**Block Parameter:** SampleTime
**Type:** character vector
**Values:** scalar | vector
**Default:** '-1'

### Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>bus</th>
<th>double</th>
<th>enumerated</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Direct Feedthrough</strong></td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Multidimensional Signals</strong></td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Variable-Size Signals | no
---|---
Zero-Crossing Detection | no

a. Supports up to 32-bit fixed-point data types.

**Tips**

- If MATLAB encounters memory issues when you log many signals in a long simulation that has many time steps, consider logging to persistent storage. When you log to persistent storage, the `Dataset` format logging data is stored in a MAT-file. Compared to logging to persistent storage, connecting a To File block to signals:
  - Is a per-signal approach that can clutter a model with several To File blocks attached to individual signals.
  - Creates a separate MAT-file for each To File block, compared to the one MAT-file that logging to persistent storage uses.

For details, see “Log Data to Persistent Storage”.

- To avoid the overhead of compressing data in real time, the To File block writes an uncompressed Version 7.3 MAT-file. To compress the data within the MAT-file, load and save the file in MATLAB. The resaved file is smaller than the original MAT-file that the To File block created, because the `Save` command compresses the data in the MAT-file.

**Extended Capabilities**

**C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

Not recommended for production code. Relates to resource limits and restrictions on speed and memory often found in embedded systems. Generated code can contain dynamic allocation and freeing of memory, recursion, additional memory overhead, and widely-varying execution times. While the code is functionally valid and generally acceptable in resource-rich environments, smaller embedded targets often cannot
support such code. Usually, blocks evolve toward being suitable for production code. Thus, blocks suitable for production code remain suitable.

Code generation for RSim target provides identical support as Simulink; all other code generation targets support only double, one-dimensional, real signals in array with time format.

To generate code for a To File block, on the Code Generation > Interface pane, select the configuration parameter “MAT-file logging” (Simulink Coder).

**HDL Code Generation**
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

This block can be used for writing simulation results to a file when used with subsystems that generate HDL code, but is not included in the hardware implementation

**PLC Code Generation**
Generate Structured Text code using Simulink® PLC Coder™.

**Fixed-Point Conversion**
Design and simulate fixed-point systems using Fixed-Point Designer™.

Supports up to 32-bit fixed-point data types.

**See Also**
From File | From Workspace | To Workspace

**Topics**
“Save Run-Time Data from Simulation”
“Convert Logged Data to Dataset Format”
“Limit Amount of Exported Data”

*Introduced before R2006a*
To Workspace

Write data to workspace

**Library:**
- Simulink / Sinks
- HDL Coder / Sinks

**Description**

The To Workspace block writes input signal data to a workspace. During simulation, the block writes data to an internal buffer. When you pause the simulation or the simulation completes, that data is written to the workspace. Data is not available until the simulation pauses or stops.

The To Workspace block typically writes data to the MATLAB base workspace. For a `sim` command in a MATLAB function, the To Workspace block sends data to the workspace of the calling function, not to the MATLAB base workspace. To send the logged data to the base workspace, use an `assignin` command in the function.

```matlab
function myfunc
    a = sim('mTest','SimulationMode','normal');
    b = a.get('simout')
    assignin('base','b',b);
end
```

The To Workspace block icon shows the name of the variable to which the data is written.

**Control Amount of Data Saved**

If you specify data logging intervals with the **Configuration Parameters > Data Import/Export > Logging intervals** parameter, the To Workspace block does not log data outside of the intervals. For example, the block logs no data if the intervals are empty (`[]`). The block stores the logged data in the variable that you specify for the **Single simulation output** parameter.

For variable-step solvers, to control the amount of data available to the To Workspace block, use the **Configuration Parameters > Data Import/Export > Additional parameters > Output options** parameter. For example, to write data at identical time points over multiple simulations, select the **Produce specified output only** option.
Block parameters also control the amount of data saved. See “Limit data points to last” on page 1-0, “Decimation” on page 1-0, and “Sample time” on page 1-0.

**Log to MAT-File**

When you enable the **MAT-file logging** parameter in **Configuration Parameters**, the To Workspace block logs its data to a MAT-file. For information about this parameter, see “MAT-file logging” (Simulink Coder).

**Ports**

**Input**

**Port_1 — Input signal**

scalar | vector | matrix | N-D array

Workspace data created from input signal. The To Workspace block can save real or complex inputs of any data type that Simulink supports, including fixed-point and enumerated data types, and bus objects.

By default, the To Workspace block treats input signals as sample-based. To have the To Workspace block treat input signals as frame-based, set:

1. **Save format** to either Array or Structure
2. **Save 2-D signals as** to 2-D array (concatenate along first dimension)

**Data Types:** single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus

**Parameters**

**Variable name — Name of variable for saved data**

simout (default) | character vector

Specify the name of the variable for the saved data.
Programmatic Use
Block Parameter: VariableName
Type: character vector
Values: character vector
Default: 'simout'

Limit data points to last — Maximum number of input samples to save
inf (default) | scalar | vector

Specify the maximum number of input samples to save. If the simulation generates more data points than the specified maximum, the simulation saves only the most recently generated samples. The default value causes the block to write all data.

Programmatic Use
Block Parameter: MaxDataPoints
Type: character vector
Values: scalar | vector
Default: 'inf'

Decimation — Decimation factor that determines when data writes
1 (default) | scalar | vector

Specify the decimation factor, n, which writes data at every nth time that the block executes. The default value causes the block to write data at every time step.

Programmatic Use
Block Parameter: Decimation
Type: character vector
Values: scalar | vector
Default: '1'

Save format — Format for saving simulation output
Timeseries (default) | Structure With Time | Structure | Array

Specify the format for saving simulation output to the workspace.

The default Timeseries format saves nonbus signals as a MATLAB timeseries object and bus signals as a structure of MATLAB timeseries objects.

The Array format saves the input as an N-dimensional array where N is one more than the number of dimensions of the input signal. For example, if the input signal is a vector, the resulting workspace array is two-dimensional. If the input signal is a matrix, then the
array is three-dimensional. How Simulink stores samples in the array depends on whether
the input signal is a scalar, vector, or matrix.

• If the input signal is a scalar or a vector, each input sample is output as a row of the
array. Suppose that the name of the output array is simout. Then, simout(1,:) corresponds to the first sample, simout(2,:) corresponds to the second sample, and so on.

• If the input signal is a matrix, time corresponds to the third dimension. Suppose again
that simout is the name of the resulting workspace array. Then, simout(:,1) is the input signal value at the first sample point, simout(:,2) is the input signal value at the second sample point, and so on.

The Structure format consists of a structure with three fields:

• time — Empty field for this format.

• signals — Structure with three fields: values, dimensions, and label. The
   values field contains the array of signal values. The dimensions field specifies the
   dimensions of the corresponding signals. The label field contains the label of the
   input line.

• blockName — Name of the To Workspace block.

The Structure With Time format is the same as Structure, except that the time field
contains a vector of simulation time hits.

If you select Array or Structure, the Save 2-D signals as parameter appears.

To read the To Workspace block output directly with a From Workspace block, use either
the Timeseries or Structure with Time format. The From Workspace block can read
sample-based data from a To Workspace block that was saved in a previous simulation.
For details, see “Comparison of Signal Loading Techniques”.

The following table shows how simulation mode support depends on the Save format
value.

<table>
<thead>
<tr>
<th>Simulation Mode</th>
<th>Timeseries</th>
<th>Array, Structure, or Structure With Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal</td>
<td>Supported</td>
<td>Supported</td>
</tr>
<tr>
<td>Accelerator</td>
<td>Supported</td>
<td>Supported only in top model, not</td>
</tr>
<tr>
<td></td>
<td></td>
<td>referenced models</td>
</tr>
<tr>
<td>Simulation Mode</td>
<td>Timeseries</td>
<td>Array, Structure, or Structure With Time</td>
</tr>
<tr>
<td>---------------------------------</td>
<td>----------------------</td>
<td>--------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Rapid Accelerator</td>
<td>Not supported</td>
<td>Supported only in top model, not referenced models</td>
</tr>
<tr>
<td>Software-in-the-loop (SIL)</td>
<td>Not supported</td>
<td>If MAT-file logging is enabled, supported only in top model, not referenced models</td>
</tr>
<tr>
<td>Processor-in-the-loop (PIL)</td>
<td>Not supported</td>
<td>If MAT-file logging is available and enabled, supported only in top model, not referenced models</td>
</tr>
<tr>
<td>External</td>
<td>Not supported</td>
<td>Supported only in top model, not referenced models</td>
</tr>
<tr>
<td>Simulink Coder Targets</td>
<td>Not supported</td>
<td>If MAT-file logging is enabled, supported only in top model, not referenced models</td>
</tr>
</tbody>
</table>

Programmatic Use

**Block Parameter:** SaveFormat

**Type:** character vector

**Values:** 'Timeseries' | 'Structure with Time' | 'Structure' | 'Array'

**Default:** 'Timeseries'

Save 2-D signal as — Format for saving 2-D signals

3-D array (concatenate along third dimension) (default) | 2-D array (concatenate along first dimension)

Specify one of these formats for saving 2-D signals to the workspace:

- **3-D array (concatenate along third dimension) (Default)**

  This setting is well-suited for sample-based signals. Data is concatenated along the third dimension. For example, 2-by-4 matrix input for 10 samples is stored as a 2-by-4-by-10 array.

- **2-D array (concatenate along first dimension)**

  This setting is well-suited for frame-based signals. Data is concatenated along the first dimension. For example, 2-by-4 matrix input for 10 samples is stored as a 20-by-4 array.
When the **Save format** is set to **Array** or **Structure**, the dimensions of the output depend on the input dimensions and the setting of the **Save 2-D signals as** parameter. The following table summarizes the output dimensions under various conditions. In the table, $K$ represents the value of the **Limit data points to last** parameter.

<table>
<thead>
<tr>
<th>Input Signal Dimensions</th>
<th>Save 2-D Signals As ...</th>
<th>Signal to Workspace Output Dimension</th>
</tr>
</thead>
<tbody>
<tr>
<td>$M$-by-$N$ matrix</td>
<td>2-D array (concatenate along first dimension)</td>
<td>$K$-by-$N$ matrix. If you set the <strong>Limit data points to last</strong> parameter to <code>inf</code>, $K$ represents the total number of samples acquired in each column by the end of simulation. This setting is equivalent to multiplying the input frame size ($M$) by the total number of $M$-by-$N$ inputs acquired by the block.</td>
</tr>
<tr>
<td>$M$-by-$N$ matrix</td>
<td>3-D array (concatenate along third dimension)</td>
<td>$M$-by-$N$-by-$K$ array. If you set the <strong>Limit data points to last</strong> parameter to <code>inf</code>, $K$ represents the total number of $M$-by-$N$ inputs acquired by the end of the simulation.</td>
</tr>
<tr>
<td>Length-$N$ unoriented vector</td>
<td>Any setting</td>
<td>$K$-by-$N$ matrix</td>
</tr>
<tr>
<td>$N$-dimensional array where $N &gt; 2$</td>
<td>Any setting</td>
<td>Array with $N + 1$ dimensions, where the size of the last dimension is equal to $K$. If you set the <strong>Limit data points to last</strong> parameter to <code>inf</code>, $K$ represents the total number of $M$-by-$N$ inputs acquired by the end of simulation</td>
</tr>
</tbody>
</table>
Dependencies

To enable the **Save 2-D signals as** parameter, set the **Save format** to **Array** or **Structure**.

**Programmatic Use**

**Block Parameter:** Save2DSignal  
**Type:** character vector  
**Values:** '2-D array (concatenate along first dimension)' | '3-D array (concatenate along third dimension)'  
**Default:** '3-D array (concatenate along third dimension)'

**Log fixed-point data as a fi object — Log fixed-point data as a fi object or double**

**off** (default) | on

By default, the To Workspace block logs fixed-point data to the MATLAB workspace as a Fixed-Point Designer fi object. If you clear this parameter, Simulink software logs fixed-point data to the workspace as double.

**Programmatic Use**

**Block Parameter:** FixptAsFi  
**Type:** character vector  
**Values:** 'off' | 'on'  
**Default:** 'off'

**Sample time — Sample period and offset**

-1 (default) | scalar | vector

Specifies the sample period and offset at which to collect data points. This parameter is useful when you are using a variable-step solver where the interval between time steps is not constant. The default value causes the block to inherit the sample time from the driving block. See “Specify Sample Time”.

**Programmatic Use**

**Block Parameter:** SampleTime  
**Type:** character vector  
**Values:** scalar | vector  
**Default:** '-1'
Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>bus</th>
<th>double</th>
<th>enumerated</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Tips

To make post-processing easier, you can convert data saved by this block to Dataset format. This conversion is useful when post processing this data with other logged data that can use Dataset format (for example, logged states). See “Convert Logged Data to Dataset Format”. You can also use signal logging with a variable-size signal exception.

Extended Capabilities

**HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

This block can be used for simulation visibility in subsystems that generate HDL code, but is not included in the hardware implementation.

**PLC Code Generation**

Generate Structured Text code using Simulink® PLC Coder™.
Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

See Also
From File | From Workspace | To File

Topics
“Export Simulation Data”
“Limit Amount of Exported Data”
“Convert Logged Data to Dataset Format”

Introduced before R2006a
Toggle Switch

Toggle parameter between two values
Library: Simulink / Dashboard

Description

The Toggle Switch block toggles the value of the connected block parameter between two values during simulation. For example, you can connect the Toggle Switch block to a Switch block in your model and change its state during simulation. Use the Toggle Switch block with other Dashboard blocks to create an interactive dashboard for your model.

Double-clicking the Toggle Switch block does not open its dialog box during simulation and when the block is selected. To edit the block's parameters, you can use the Property Inspector, or you can right-click the block and select Block Parameters from the context menu.

Connecting Dashboard Blocks

Dashboard blocks do not use ports to make connections. To connect Dashboard blocks to variables and block parameters in your model, use connect mode. Connect mode facilitates the process of connecting Dashboard blocks in your model, especially when you want to connect multiple blocks at once. If you only want to connect a single Dashboard block, you can also use the Connection table in the block dialog box to make the connection.

Tip To facilitate understanding and debugging your model, you can modify Dashboard block connections in your model during normal and accelerator mode simulations.
**Note** Dashboard blocks cannot connect to variables until you update your model diagram. To connect Dashboard blocks to variables or modify variable values between opening your model and running a simulation, update your model diagram using **Ctrl+D**.

To enter connect mode, click the **Connect** button that appears above your unconnected Dashboard block when you pause on it.

In connect mode, when you select one or more signals or blocks, a list of parameters and signals available for connection appears. Select a signal or parameter from the list to connect the selected Dashboard block. To connect another Dashboard block, pause on the block and click the **Connect** button above it. Then, make a selection of signals and blocks in your model, and choose a signal or parameter to connect.
When you finish connecting the Dashboard blocks in your model, click the **Exit** button in the upper-right of the canvas to exit connect mode.

**Tip** You can hide the message shown on unconnected blocks using the `set_param` function with the `ShowInitialText` block parameter.

**Parameter Logging**

Tunable parameters connected to Dashboard blocks are logged to the Simulation Data Inspector, where you can view parameter values along with logged signal data. You can access logged parameter data in the MATLAB workspace by exporting the parameter data from the Simulation Data Inspector UI or by using the `Simulink.sdi.exportRun` function. For more information about exporting data with the Simulation Data Inspector UI, see “Export Data from the Simulation Data Inspector”. The parameter data is stored in a `Simulink.SimulationData.Parameter` object, accessible as an element in the exported `Simulink.SimulationData.Dataset`. 

1-2128
Limitations

- Dashboard blocks can only connect to real scalar signals.
- You cannot use the Connection table to connect a Dashboard block to a block that is commented out. When you connect a Dashboard block to a commented block using connect mode, the Dashboard block does not display the connected value until the you uncomment the block.
- Dashboard blocks cannot connect to signals inside referenced models.
- When you simulate a model hierarchy, Dashboard blocks inside referenced models do not update.
- Dashboard blocks do not support rapid accelerator simulation.
- Some signals are not available during accelerator mode simulations. To view such signals on a Dashboard block, mark the signal for logging.
- Parameters specified by indexing a variable array do not appear in the Connection table. For example, a block parameter defined as engine(1) using the variable engine does not appear in the table.

To access the parameter in the Connection table, assign the indexed value to a scalar variable, such as engine_1. Then, use the scalar variable to define the block parameter.

- When you connect a Dashboard block to a variable or parameter during simulation, the data for that variable or parameter does not get logged to the Simulation Data Inspector. To log variable and parameter data to the Simulation Data Inspector, connect the Dashboard block to the variable or parameter prior to simulation.
- When you simulate a model in external mode with the Default parameter behavior set to Inlined, Dashboard blocks can appear to change parameter and variable values. However, the change does not propagate to the simulation. For example, the Gain block displays changes made to its Gain parameter using the Dashboard blocks, but the Gain value used in the simulation does not change.

Parameters

Connection — Select a variable or block parameter to connect
variable and parameter connection options

Select the variable or block parameter to control using the Connection table. Populate the Connection table by selecting one or more blocks in your model. Select the radio
button next to the variable or parameter you want to control, and click **Apply**. To facilitate understanding and debugging your model, you can connect Dashboard blocks to variables and parameters in your model during simulation.

**Note** To see workspace variables in the connection table, update the model diagram using **Ctrl+D**.

**Programmatic Use**

To programmatically connect a Dashboard block to a tunable parameter or a variable, use a `Simulink.HMI.ParamSourceInfo` object. The `Simulink.HMI.ParamSourceInfo` object contains four properties. Some of the properties apply to connecting Dashboard blocks to parameters, and some apply to connecting Dashboard blocks to variables. Not all fields have a value for a connection because a given Dashboard block connects to either a parameter or a variable.

**Block Parameter:** Binding  
**Type:** `Simulink.HMI.ParamSourceInfo`  
**Default:** `[ ]`

**States — Pair values and labels**

Scalar and character vector

Pairs of values to assign to the connected variable or parameter and text to display on the block. Switches have two states — **Top** and **Bottom** — one corresponding to each switch position. Each state contains a **Value** and a **Label**.

- **Value** — Value to assign to the connected variable or parameter when the switch is in the corresponding position.
- **Label** — Text to display on the block for the corresponding position.

This table describes the default configuration for the block.

**States**

<table>
<thead>
<tr>
<th>Position</th>
<th>State Value</th>
<th>State Label</th>
</tr>
</thead>
<tbody>
<tr>
<td>Top</td>
<td>0</td>
<td>on</td>
</tr>
<tr>
<td>Bottom</td>
<td>1</td>
<td>off</td>
</tr>
</tbody>
</table>
**Programmatic Use**

To configure the **States** for the block programmatically, specify the value of the **States** parameter as a structure array containing two elements with fields:

- **Value** — Scalar double value for the state.
- **Label** — String or character array to use as the label for the switch position.

```matlab
topState.Value = 0;
topState.Label = 'Off';
bottomState.Value = 1;
bottomState.Label = 'On';
switchStates = [topState bottomState];
```

**Block Parameter:** States  
**Type:** two element array of structures

**Label — Block label position**  
**Top** (default) | **Bottom** | **Hide**

Position of the block label. When the block is connected to an element in the model, the label is the name of the connected element.

**Programmatic Use**  
**Block Parameter:** LabelPosition  
**Type:** character vector  
**Values:** 'Top' | 'Bottom' | 'Hide'  
**Default:** 'Top'

**Block Characteristics**

<table>
<thead>
<tr>
<th>Data Types</th>
<th>double</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>no</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Extended Capabilities

**C/C++ Code Generation**
Generate C and C++ code using Simulink® Coder™.

Dashboard blocks are ignored for code generation.

**See Also**
Rocker Switch | Rotary Switch | Slider Switch

**Topics**
- “Tune and Visualize Your Model with Dashboard Blocks”
- “Decide How to Visualize Simulation Data”

**Introduced in R2015a**
To String

Convert input signal to string signal

Library: Simulink / String

Description

The To String block creates a string signal from an input signal. For example, consider using this signal to convert a logical value 1 or 0 to its string equivalent "false" or "true".

When a MinGW compiler compiles code generated from the block, running the compiled code may produce nonstandard results for floating-point inputs. For example, a numeric input of 501.987 returns the string "5.019870e+002" instead of the expected string "5.019870e+02".

Ports

Input

Port_1 — Input signal
scalar

Input signal, specified as a scalar.

Output

Port_1 — Output string
scalar

Output string, specified as a scalar. This block returns the output as a string, surrounded by double quotes.
• If the input is a Boolean, the output is a logical value (1 or 0) and the block returns its textual equivalent (true or false).

• If the input is a numeric data type, such as an integer, single, double, or fixed point, the block returns the number as a string. For example, an input of 1 converts to "1" and an input of 0 converts to "0".

**Note** The output string might not contain all the digits of the numeric value from the input port.

Data Types: string

### Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>double</th>
<th>enumerated</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
<th>string</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Extended Capabilities

**C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.
Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

See Also
ASCII to String | Compose String | Scan String | String Compare | String Concatenate | String Constant | String Find | String Length | String To ASCII | String To Enum | String to Double | String to Single | Substring

Topics
“Get Text Following a Keyword”
“Simulink Strings”

Introduced in R2018a
Transfer Fcn

Model linear system by transfer function
Library: Simulink / Continuous

Description

The Transfer Fcn block models a linear system by a transfer function of the Laplace-domain variable \( s \). The block can model single-input single-output (SISO) and single-input multiple-output (SIMO) systems.

Conditions for Using This Block

The Transfer Fcn block assumes the following conditions:

- The transfer function has the form

\[
H(s) = \frac{y(s)}{u(s)} = \frac{\text{num}(s)}{\text{den}(s)} = \frac{\text{num}(1)s^{nn-1} + \text{num}(2)s^{nn-2} + \ldots + \text{num}(nn)}{\text{den}(1)s^{nd-1} + \text{den}(2)s^{nd-2} + \ldots + \text{den}(nd)},
\]

where \( u \) and \( y \) are the system input and outputs, respectively, \( nn \) and \( nd \) are the number of numerator and denominator coefficients, respectively. \( \text{num}(s) \) and \( \text{den}(s) \) contain the coefficients of the numerator and denominator in descending powers of \( s \).

- The order of the denominator must be greater than or equal to the order of the numerator.
- For a multiple-output system, all transfer functions have the same denominator and all numerators have the same order.

Modeling a Single-Output System

For a single-output system, the input and output of the block are scalar time-domain signals. To model this system:

1. Enter a vector for the numerator coefficients of the transfer function in the Numerator coefficients field.
2 Enter a vector for the denominator coefficients of the transfer function in the **Denominator coefficients** field.

### Modeling a Multiple-Output System

For a multiple-output system, the block input is a scalar and the output is a vector, where each element is an output of the system. To model this system:

1 Enter a matrix in the **Numerator coefficients** field.

   Each row of this matrix contains the numerator coefficients of a transfer function that determines one of the block outputs.

2 Enter a vector of the denominator coefficients common to all transfer functions of the system in the **Denominator coefficients** field.

### Specifying Initial Conditions

A transfer function describes the relationship between input and output in Laplace (frequency) domain. Specifically, it is defined as the Laplace transform of the response (output) of a system with zero initial conditions to an impulse input.

Operations like multiplication and division of transfer functions rely on zero initial state. For example, you can decompose a single complicated transfer function into a series of simpler transfer functions. Apply them sequentially to get a response equivalent to that of the original transfer function. This will not be correct if one of the transfer functions assumes a non-zero initial state. Furthermore, a transfer function has infinitely many time domain realizations, most of whose states do not have any physical meaning.

For these reasons, Simulink presets the initial conditions of the Transfer Fcn block to zero. To specify initial conditions for a given transfer function, convert the transfer function to its controllable, canonical state-space realization using `tf2ss`. Then, use the State-Space block. The `tf2ss` utility provides the A, B, C, and D matrices for the system.

For more information, type `help tf2ss` or see the Control System Toolbox™ documentation.

### Transfer Function Display on the Block

The Transfer Fcn block displays the transfer function depending on how you specify the numerator and denominator parameters.
• If you specify each parameter as an expression or a vector, the block shows the transfer function with the specified coefficients and powers of \( s \). If you specify a variable in parentheses, the block evaluates the variable.

For example, if you specify **Numerator coefficients** as \([3, 2, 1]\) and **Denominator coefficients** as \((\text{den})\), where \(\text{den}\) is \([7, 5, 3, 1]\), the block looks like this:

\[
\frac{3s^2+2s+1}{7s^3+5s^2+3s+1}
\]

• If you specify each parameter as a variable, the block shows the variable name followed by \((s)\).

For example, if you specify **Numerator coefficients** as \(\text{num}\) and **Denominator coefficients** as \(\text{den}\), the block looks like this:

\[
\frac{\text{num}(s)}{\text{den}(s)}
\]

### Ports

#### Input

**Port 1 — Input signal**

**scalar**

Input signal, specified as a scalar with data type **double**.

Data Types: **double**
Output

Port_1 — Output signal
scalar | vector

Output signal, provided as a scalar or vector with data type double.

• For a single-output system, the input and output of the block are scalar time-domain signals.
• For a multiple-output system, the input is a scalar, and the output is a vector, where each element is an output of the system.

Data Types: double

Parameters

Numerator coefficients — Vector or matrix of numerator coefficients
[1] (default) | vector | matrix

Define the numerator coefficients of the transfer function.

• For a single-output system, enter a vector for the numerator coefficients of the transfer function.
• For a multiple-output system, enter a matrix. Each row of this matrix contains the numerator coefficients of a transfer function that determines one of the block outputs.

Programmatic Use
Block Parameter: Numerator
Type: character vector, string
Values: vector | matrix
Default: ' [1]'

Denominator coefficients — Row vector of denominator coefficients
[1 1] (default) | vector

Define the row vector of denominator coefficients.

• For a single-output system, enter a vector for the denominator coefficients of the transfer function.
• For a multiple-output system, enter a vector containing the denominator coefficients common to all transfer functions of the system.

**Programmatic Use**

**Block Parameter:** Denominator  
**Type:** character vector, string  
**Values:** vector  
**Default:** '[1 1]'

**Absolute tolerance — Absolute tolerance for computing block states**  
*auto* (default) | scalar | vector

Absolute tolerance for computing block states, specified as a positive, real-valued, scalar or vector. To inherit the absolute tolerance from the Configuration Parameters, specify auto or -1.

• If you enter a real scalar, then that value overrides the absolute tolerance in the Configuration Parameters dialog box for computing all block states.

• If you enter a real vector, then the dimension of that vector must match the dimension of the continuous states in the block. These values override the absolute tolerance in the Configuration Parameters dialog box.

• If you enter auto or -1, then Simulink uses the absolute tolerance value in the Configuration Parameters dialog box (see “Solver Pane”) to compute block states.

**Programmatic Use**

**Block Parameter:** AbsoluteTolerance  
**Type:** character vector, string  
**Values:** 'auto' | '-1' | any positive real-valued scalar or vector  
**Default:** 'auto'

**State Name (e.g., 'position') — Assign unique name to each state**  
' ' (default) | 'position' | {'a', 'b', 'c'} | a | ...

Assign a unique name to each state. If this field is blank (' '), no name assignment occurs.

• To assign a name to a single state, enter the name between quotes, for example, 'position'.

• To assign names to multiple states, enter a comma-delimited list surrounded by braces, for example, {'a', 'b', 'c'}. Each name must be unique.
• To assign state names with a variable in the MATLAB workspace, enter the variable without quotes. A variable can be a character vector, string, cell array, or structure.

**Limitations**

• The state names apply only to the selected block.
• The number of states must divide evenly among the number of state names.
• You can specify fewer names than states, but you cannot specify more names than states.

For example, you can specify two names in a system with four states. The first name applies to the first two states and the second name to the last two states.

**Programmatic Use**

**Block Parameter:** ContinuousStateAttributes

**Type:** character vector; string

**Values:** ' ' | user-defined

**Default:** ' '

---

**Block Characteristics**

<table>
<thead>
<tr>
<th>Data Types</th>
<th>double</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>yes</td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>no</td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
</tr>
</tbody>
</table>

---
Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

Not recommended for production-quality code. Relates to resource limits and restrictions on speed and memory often found in embedded systems. The code generated can contain dynamic allocation and freeing of memory, recursion, additional memory overhead, and widely-varying execution times. While the code is functionally valid and generally acceptable in resource-rich environments, smaller embedded targets often cannot support such code.

In general, consider using the Simulink Model Discretizer to map continuous blocks into discrete equivalents that support production code generation. To start the Model Discretizer, select Analysis > Control Design > Model Discretizer. One exception is the Second-Order Integrator block because, for this block, the Model Discretizer produces an approximate discretization.

See Also
Discrete Transfer Fcn | State-Space

Topics
“State”

Introduced before R2006a
Transfer Fcn Direct Form II

Implement Direct Form II realization of transfer function

Library: Simulink / Additional Math & Discrete / Additional
Discrete

Description

The Transfer Fcn Direct Form II block implements a Direct Form II realization of the transfer function that the Numerator coefficients and Denominator coefficients excluding lead parameters specify. The block supports only single input-single output (SISO) transfer functions.

The block automatically selects the data types and scalings of the output, the coefficients, and any temporary variables.

Ports

Input

Port_1 — Input signal
scalar | vector

Input signal, specified as a scalar or vector.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

Output

Port_1 — Output signal
scalar | vector

Output signal, specified as a scalar or vector.
Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

Parameters

Numerator coefficients — Numerator coefficients
[0.2 0.3 0.2] (default) | vector

Specify the numerator coefficients as a vector.

Programmatic Use
Block Parameter: NumCoefVec
Type: character vector
Values: vector
Default: '

Denominator coefficients excluding lead — Denominator coefficient
[-0.9 0.6] (default) | vector

Specify the denominator coefficients, excluding the leading coefficient, which must be 1.0.

Programmatic Use
Block Parameter: DenCoeffVec
Type: character vector
Values: vector
Default: '

Initial condition — Initial condition
0.0 (default) | scalar

Specify the initial condition as a scalar.

Programmatic Use
Block Parameter: vinit
Type: character vector
Values: scalar
Default: '0.0'

Integer rounding mode — Rounding mode for fixed-point operations
Floor (default) | Ceiling | Convergent | Nearest | Round | Simplest | Zero
Specify the rounding mode for fixed-point operations. For more information, see “Rounding” (Fixed-Point Designer).

Programmatic Use
Block Parameter: RndMeth
Type: character vector
Values: 'Ceiling' | 'Convergent' | 'Floor' | 'Nearest' | 'Round' | 'Simplest' | 'Zero'
Default: 'Floor'

Saturate to max or min when overflows occur — Method of overflow action
off (default) | on

When you select this check box, overflows saturate to the maximum or minimum value that the data type can represent. Otherwise, overflows wrap.

When you select this check box, saturation applies to every internal operation on the block, not just the output or result. In general, the code generation process can detect when overflow is not possible. In this case, the code generator does not produce saturation code.

Programmatic Use
Block Parameter: DoSatur
Type: character vector
Values: 'off' | 'on'
Default: 'off'

Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>double</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Extended Capabilities

### C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

The code generator does not explicitly group primitive blocks that constitute a nonatomic masked subsystem block in the generated code. This flexibility allows for more efficient code generation. In certain cases, you can achieve grouping by configuring the masked subsystem block to execute as an atomic unit by selecting the **Treat as atomic unit** option.

### PLC Code Generation
Generate Structured Text code using Simulink® PLC Coder™.

### Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

### See Also
Discrete Transfer Fcn | Transfer Fcn Direct Form II Time Varying

### Topics
“Fixed-Point Numbers”

**Introduced before R2006a**
Transfer Fcn Direct Form II Time Varying

Implement time varying Direct Form II realization of transfer function

Library: Simulink / Additional Math & Discrete / Additional
         Discrete

Description

The Transfer Fcn Direct Form II Time Varying block implements a Direct Form II realization of the specified transfer function. The block supports only single input-single output (SISO) transfer functions.

The input signal labeled Den No Lead contains the denominator coefficients of the transfer function. The full denominator has a leading coefficient of one, but it is excluded from the input signal. For example, to use a denominator of \([1 \ -1.7 \ 0.72]\), specify a signal with the value \([-1.7 \ 0.72]\). The input signal labeled Num contains the numerator coefficients. The data types of the numerator and denominator coefficients can be different, but the length of the numerator vector and the full denominator vector must be the same. Pad the numerator vector with zeros, if needed.

The block automatically selects the data types and scalings of the output, the coefficients, and any temporary variables.

Ports

Input

\(u\) — Input signal
scalar | vector

Input signal, specified as a scalar or vector.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point

1-2147
**Num — Numerator coefficients**

Scalar | Vector

Numerator coefficients of the transfer function, specified as a scalar or vector.

**Dependencies**

The data types of the numerator and denominator coefficients can be different, but the length of the numerator vector and the full denominator vector must be the same. Pad the numerator vector with zeros, if needed.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point

**Den No Lead — Denominator coefficients**

Scalar | Vector

Denominator coefficients of the transfer function, specified as a scalar or vector, without the leading coefficient of one. The full denominator has a leading coefficient of one, but it is excluded from the input signal.

Example: For a denominator coefficient of \([1 \ -1.7 \ 0.72]\), specify a signal with value \([-1.7 \ 0.72]\).

**Dependencies**

The data types of the numerator and denominator coefficients can be different, but the length of the numerator vector and the full denominator vector must be the same.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point

**Output**

**y — Output signal**

Scalar | Vector

Output signal, specified as a scalar or vector.

The block automatically selects the data types and scalings of the output and any temporary variables.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus
Parameters

**Initial condition — Initial condition**
0.0 (default) | scalar

Specify the initial condition as a scalar.

**Programmatic Use**

**Block Parameter:** vinit  
**Type:** character vector  
**Values:** scalar  
**Default:** '0.0'

**Integer rounding mode — Rounding mode for fixed-point operations**
Floor (default) | Ceiling | Convergent | Nearest | Round | Simplest | Zero

Specify the rounding mode for fixed-point operations. For more information, see “Rounding” (Fixed-Point Designer).

**Programmatic Use**

**Block Parameter:** RndMeth  
**Type:** character vector  
**Values:** 'Ceiling' | 'Convergent' | 'Floor' | 'Nearest' | 'Round' | 'Simplest' | 'Zero'  
**Default:** 'Floor'

**Saturate to max or min when overflows occur — Method of overflow action**
off (default) | on

When you select this check box, overflows saturate to the maximum or minimum value that the data type can represent. Otherwise, overflows wrap.

When you select this check box, saturation applies to every internal operation on the block, not just the output or result. In general, the code generation process can detect when overflow is not possible. In this case, the code generator does not produce saturation code.

**Programmatic Use**

**Block Parameter:** DoSatur  
**Type:** character vector  
**Values:** 'off' | 'on'  
**Default:** 'off'
Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean(^a)</th>
<th>double</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\(^a\) This block is not recommended for use with Boolean signals.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

The code generator does not explicitly group primitive blocks that constitute a nonatomic masked subsystem block in the generated code. This flexibility allows for more efficient code generation. In certain cases, you can achieve grouping by configuring the masked subsystem block to execute as an atomic unit by selecting the Treat as atomic unit option.

PLC Code Generation
Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.
See Also
Discrete Transfer Fcn | Transfer Fcn Direct Form II

Topics
“Fixed-Point Numbers”

Introduced before R2006a
Transfer Fcn First Order

Implement discrete-time first order transfer function
Library: Simulink / Discrete

Description

The Transfer Fcn First Order block implements a discrete-time first order transfer function of the input. The transfer function has a unity DC gain.

Ports

Input

Port_1 — Input signal
scalar | vector

Input signal to the first order transfer function algorithm.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point

Output

Port_1 — Transfer function output signal
scalar | vector

Output signal that is the discrete-time first order transfer function of the input with a unity DC gain.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point
Parameters

Pole (in Z plane) — Pole
0.95 (default) | scalar

Specify the pole.

Programmatic Use
Block Parameter: PoleZ
Type: character vector
Value: real scalar
Default: ’0.95’

Initial condition for previous output — Initial condition for previous output
0.0 (default) | scalar

Specify the initial condition for the previous output.

Programmatic Use
Block Parameter: ICPrevOutput
Type: character vector
Value: real scalar
Default: ’0.0’

Integer rounding mode — Rounding mode for fixed-point operations
Floor (default) | Ceiling | Convergent | Nearest | Round | Simplest | Zero

Specify the rounding mode for fixed-point operations. For more information, see “Rounding” (Fixed-Point Designer).

Block parameters always round to the nearest representable value. To control the rounding of a block parameter, enter an expression using a MATLAB rounding function into the mask field.

Programmatic Use
Block Parameter: RndMeth
Type: character vector
Values: ’Ceiling’ | ’Convergent’ | ’Floor’ | ’Nearest’ | ’Round’ | ’Simplest’ | ’Zero’
Default: ’Floor’
**Saturate to max or min when overflows occur — Method of overflow action**

off (default) | on

When you select this check box, overflows saturate to the maximum or minimum value that the data type can represent. Otherwise, overflows wrap.

When you select this check box, saturation applies to every internal operation on the block, not just the output or result. In general, the code generation process can detect when overflow is not possible. In this case, the code generator does not produce saturation code.

**Programmatic Use**

**Block Parameter:** DoSatur

**Type:** character vector

**Values:** 'off' | 'on'

**Default:** 'off'

### Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>double</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Extended Capabilities

**C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.
PLC Code Generation
Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

See Also
Transfer Fcn | Transfer Fcn Lead or Lag

Introduced before R2006a
Transfer Fcn Lead or Lag

Implement discrete-time lead or lag compensator

Library: Simulink / Discrete

Description

The Transfer Fcn Lead or Lag block implements a discrete-time lead or lag compensator of the input. The instantaneous gain of the compensator is 1, and the DC gain is equal to \((1 - z)/(1 - p)\), where \(z\) is the zero and \(p\) is the pole of the compensator.

The block implements a lead compensator when \(0 < z < p < 1\), and implements a lag compensator when \(0 < p < z < 1\).

Ports

Input

Port_1 — Input signal
scalar | vector

Input signal that the block applies the discrete-time lead or lag compensation to.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

Output

Port_1 — Output signal
scalar | vector

Output signal that is discrete-time lead or lag compensation of the input signal.
Parameters

Pole of compensator (in Z plane) — Pole

Specify the pole of the compensator.

Programmatic Use

Block Parameter: PoleZ

Type: character vector

Value: real scalar

Default: '0.95'

Zero of compensator (in Z plane) — Zero of compensator

Specify the zero of compensator in the Z plane.

Programmatic Use

Block Parameter: ZeroZ

Type: character vector

Value: real scalar

Default: '0.75'

Initial condition for previous output — Initial condition for previous output

Specify the initial condition for the previous output.

Programmatic Use

Block Parameter: ICPrevOutput

Type: character vector

Value: real scalar

Default: '0.0'

Initial condition for previous input — Initial condition for previous input

Specify the initial condition for the previous input.

Programmatic Use

Block Parameter: ICPrevInput

Type: character vector

Value: real scalar

Default: '0.0'
Specify the initial condition for the previous input.

**Programmatic Use**
**Block Parameter**: ICPrevInput
**Type**: character vector
**Value**: real scalar
**Default**: '0.0'

**Initial condition for previous output** — Initial condition for previous output
0.0 (default) | scalar

Specify the initial condition for the previous output.

**Programmatic Use**
**Block Parameter**: ICPrevOutput
**Type**: character vector
**Value**: real scalar
**Default**: '0.0'

**Integer rounding mode** — Rounding mode for fixed-point operations
Floor (default) | Ceiling | Convergent | Nearest | Round | Simplest | Zero

Specify the rounding mode for fixed-point operations. For more information, see “Rounding” (Fixed-Point Designer).

Block parameters always round to the nearest representable value. To control the rounding of a block parameter, enter an expression using a MATLAB rounding function into the mask field.

**Programmatic Use**
**Block Parameter**: RndMeth
**Type**: character vector
**Values**: 'Ceiling' | 'Convergent' | 'Floor' | 'Nearest' | 'Round' | 'Simplest' | 'Zero'
**Default**: 'Floor'

**Saturate to max or min when overflows occur** — Method of overflow action
off (default) | on

When you select this check box, overflows saturate to the maximum or minimum value that the data type can represent. Otherwise, overflows wrap.
When you select this check box, saturation applies to every internal operation on the block, not just the output or result. In general, the code generation process can detect when overflow is not possible. In this case, the code generator does not produce saturation code.

**Programmatic Use**

**Block Parameter:** DoSatur  
**Type:** character vector  
**Values:** 'off' | 'on'  
**Default:** 'off'

**Block Characteristics**

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>double</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---
a. This block is not recommended for use with Boolean signals.

**Extended Capabilities**

**C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

**PLC Code Generation**

Generate Structured Text code using Simulink® PLC Coder™.
Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

See Also
Transfer Fcn | Transfer Fcn First Order

Introduced before R2006a
Transfer Fcn Real Zero

Implement discrete-time transfer function that has real zero and no pole

Library: Simulink / Discrete

Description

The Transfer Fcn Real Zero block implements a discrete-time transfer function that has a real zero and effectively no pole.

Ports

Input

Port_1 — Input signal
scalar | vector

Input signal to the discrete-time transfer function algorithm.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

Output

Port_1 — Output signal
scalar | vector

Output signal that is the discrete-time transfer function with a real zero and effectively no pole of the input signal.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point
Parameters

Zero (in Z plane) — Zero
0.75 (default) | scalar

Specify the zero in the Z plane.

Programmatic Use

Block Parameter: ZeroZ
Type: character vector
Value: real scalar
Default: '0.75'

Initial condition for previous input — Initial condition for previous input
0.0 (default) | scalar

Specify the initial condition for the previous input.

Programmatic Use

Block Parameter: ICPrevInput
Type: character vector
Value: real scalar
Default: '0.0'

Input processing — Specify sample- or frame-based processing
Elements as channels (sample based) (default) | Columns as channels (frame based)

Specify whether the block performs sample- or frame-based processing:

- Columns as channels (frame based) — Treat each column of the input as a separate channel (frame-based processing).

Note Frame-based processing requires a DSP System Toolbox license.

For more information, see “Sample- and Frame-Based Concepts” (DSP System Toolbox).

- Elements as channels (sample based) — Treat each element of the input as a separate channel (sample-based processing).
Use **Input processing** to specify whether the block performs sample- or frame-based processing. The block accepts frame-based signals for the input \( u \). All other input signals must be sample-based.

<table>
<thead>
<tr>
<th>Input Signal ( u )</th>
<th>Input Processing Mode</th>
<th>Block Works?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sample based</td>
<td>Sample based</td>
<td>Yes</td>
</tr>
<tr>
<td>Frame based</td>
<td></td>
<td>No, produces an error</td>
</tr>
<tr>
<td>Sample based</td>
<td>Frame based</td>
<td>Yes</td>
</tr>
<tr>
<td>Frame based</td>
<td></td>
<td>Yes</td>
</tr>
</tbody>
</table>

For more information about these two processing modes, see “Sample- and Frame-Based Concepts” (DSP System Toolbox).

**Programmatic Use**

**Block Parameter:** InputProcessing  
**Type:** character vector  
**Values:** 'Columns as channels (frame based)' | 'Elements as channels (sample based)'  
**Default:** 'Elements as channels (sample based)'

**Integer rounding mode — Rounding mode for fixed-point operations**

Floor (default) | Ceiling | Convergent | Nearest | Round | Simplest | Zero

Specify the rounding mode for fixed-point operations. For more information, see “Rounding” (Fixed-Point Designer).

Block parameters always round to the nearest representable value. To control the rounding of a block parameter, enter an expression using a MATLAB rounding function into the mask field.

**Programmatic Use**

**Block Parameter:** RndMeth  
**Type:** character vector  
**Values:** 'Ceiling' | 'Convergent' | 'Floor' | 'Nearest' | 'Round' | 'Simplest' | 'Zero'  
**Default:** 'Floor'

**Saturate to max or min when overflows occur — Method of overflow action**

off (default) | on
When you select this check box, overflows saturate to the maximum or minimum value that the data type can represent. Otherwise, overflows wrap.

When you select this check box, saturation applies to every internal operation on the block, not just the output or result. In general, the code generation process can detect when overflow is not possible. In this case, the code generator does not produce saturation code.

**Programmatic Use**

**Block Parameter:** `DoSatur`

**Type:** character vector

**Values:** 'off' | 'on'

**Default:** 'off'

### Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Booleana</th>
<th>double</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Direct Feedthrough</strong></td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Multidimensional Signals</strong></td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Variable-Size Signals</strong></td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Zero-Crossing Detection</strong></td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

a. This block is not recommended for use with Boolean signals.

### Extended Capabilities

#### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.
**PLC Code Generation**
Generate Structured Text code using Simulink® PLC Coder™.

**Fixed-Point Conversion**
Design and simulate fixed-point systems using Fixed-Point Designer™.

**See Also**
Transfer Fcn | Transfer Fcn Lead or Lag

*Introduced before R2006a*
Transport Delay

Delay input by given amount of time
Library: Simulink / Continuous

Description

The Transport Delay block delays the input by a specified amount of time. You can use this block to simulate a time delay. The input to this block should be a continuous signal.

At the start of simulation, the block outputs the Initial output parameter until the simulation time exceeds the Time delay parameter. Then, the block begins generating the delayed input. During simulation, the block stores input points and simulation times in a buffer. You specify this size with the Initial buffer size parameter.

When you want output at a time that does not correspond to times of the stored input values, the block interpolates linearly between points. When the delay is smaller than the step size, the block extrapolates from the last output point, which can produce inaccurate results. Because the block does not have direct feedthrough, it cannot use the current input to calculate an output value. For example, consider a fixed-step simulation with a step size of 1 and the current time at t = 5. If the delay is 0.5, the block must generate a point at t = 4.5. Because the most recent stored time value is at t = 4, the block performs forward extrapolation.

The Transport Delay block does not interpolate discrete signals. Instead, the block returns the discrete value at the required time.

This block differs from the Unit Delay block, which delays and holds the output on sample hits only.

Tip Avoid using linmod to linearize a model that contains a Transport Delay block. For more information, see “Linearizing Models”.

1-2166
Ports

Input

Port_1 — Input signal
scalar | vector | matrix

Input signal to delay, specified as a scalar, vector, or matrix.
Data Types: double

Output

Port_1 — Delayed signal
scalar | vector | matrix

Input signal, delayed by specified amount of time. Output has the same dimensions and data type as the input signal.
Data Types: double

Parameters

Time delay — Time delay
1 (default) | scalar | vector | matrix

Specify the amount of simulation time to delay the input signal before propagation to the output as a nonnegative scalar, vector, or matrix.

Programmatic Use:
Block Parameter: DelayTime
Type: character vector, string
Values: nonnegative scalar, vector, or matrix
Default: '1'

Initial output — Initial output value
0 (default) | scalar | vector | matrix

Specify the output that the block generates until the simulation time first exceeds the time delay input as a scalar, vector, or matrix.
Limitations

The initial output of this block cannot be inf or NaN.

A Run-to-run tunable parameter cannot be changed during a simulation run time. However, changing it before a simulation begins does not cause Accelerator or Rapid Accelerator to regenerate code.

Programmatic Use

Block Parameter: InitialOutput
Type: character vector, string
Values: scalar | vector | matrix
Default: '0'

Initial buffer size — Initial memory allocation
1024 (default) | positive integer scalar

Define the initial memory allocation for the number of input points to store.

- If the number of input points exceeds the initial buffer size, the block allocates additional memory.
- After simulation ends, a message shows the total buffer size needed.

Tips

- Because allocating memory slows down simulation, choose this value carefully if simulation speed is an issue.
- For long time delays, this block can use a large amount of memory, particularly for dimensionalized input.

Programmatic Use

Block Parameter: BufferSize
Type: character vector, string
Value: positive integer scalar
Default: '1024'

Use fixed buffer size — Use fixed-size buffer
off (default) | on

Select this check box to use a fixed-size buffer to save input data from previous time steps.
The **Initial buffer size** parameter specifies the size of the buffer. If the buffer is full, new data replaces data already in the buffer. Simulink software uses linear extrapolation to estimate output values that are not in the buffer.

**Note** If you have a Simulink Coder license, ERT or GRT code generation uses a fixed-size buffer even if you do not select this check box.

**Tips**

- If the input data is linear, selecting this check box can save memory.
- If the input data is nonlinear, do not select this check box. Doing so can yield inaccurate results.

**Programmatic Use**

**Block Parameter:** FixedBuffer  
**Type:** character vector, string  
**Value:** 'off' | 'on'  
**Default:** 'off'

**Direct feedthrough of input during linearization — Enable direct feedthrough**

off (default) | on

Cause the block to output its input during linearization and trim, which sets the block mode to direct feedthrough.

**Tips**

- Selecting this check box can cause a change in the ordering of states in the model when you use the functions linmod, dlinmod, or trim. To extract this new state ordering:
  1. Compile the model using the following command, where model is the name of the Simulink model.

     ```
     [sizes, x0, x_str] = model([],[],[],'lincompile');
     ```

  2. Terminate the compilation with the following command.

     ```
     model([],[],[],'term');
     ```

- The output argument `x_str`, which is a cell array of the states in the Simulink model, contains the new state ordering. When you pass a vector of states as input to the
linmod, dlinmod, or trim functions, the state vector must use this new state ordering.

**Programmatic Use**
**Block Parameter:** TransDelayFeedthrough  
**Type:** character vector, string  
**Value:** 'off' | 'on'  
**Default:** 'off'

**Pade order (for linearization) — Order of Pade approximation**  
0 (default) | scalar | vector | matrix

Set the order of the Pade approximation for linearization routines as a scalar, vector, or matrix of nonnegative integers.

- The default value is 0, which results in a unity gain with no dynamic states.
- Setting the order to a positive integer \( n \) adds \( n \) states to your model, but results in a more accurate linear model of the transport delay.

**Programmatic Use**  
**Block Parameter:** PadeOrder  
**Type:** character vector, string  
**Values:** scalar | vector | matrix  
**Default:** '0'

**Block Characteristics**

<table>
<thead>
<tr>
<th>Data Types</th>
<th>double</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>yes</td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>no</td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
</tr>
</tbody>
</table>
Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

Not recommended for production-quality code. Relates to resource limits and restrictions on speed and memory often found in embedded systems. The code generated can contain dynamic allocation and freeing of memory, recursion, additional memory overhead, and widely-varying execution times. While the code is functionally valid and generally acceptable in resource-rich environments, smaller embedded targets often cannot support such code.

In general, consider using the Simulink Model Discretizer to map continuous blocks into discrete equivalents that support production code generation. To start the Model Discretizer, select Analysis > Control Design > Model Discretizer. One exception is the Second-Order Integrator block because, for this block, the Model Discretizer produces an approximate discretization.

See Also
Unit Delay | Variable Time Delay

Topics
“Model Discretizer”

Introduced before R2006a
Trigger

Add trigger or function port to subsystem or model

Library:  
- Simulink / Ports & Subsystems
- HDL Coder / Ports & Subsystems

Description

The Trigger block adds an external signal to control the execution of a subsystem or a model. To enable this functionality, add this block to a Subsystem block or at the root level of a model that is referenced in a Model block.

Then, configure the Trigger block to execute a subsystem or model:

- Once at each time step, when the value of the control signal changes in a way that you specify.
- Multiple times during a time step, when the control signal is a function-call event from a Stateflow chart, Function-Call Generator block, or S-Function block.

Ports

Output

Control signal — External control signal to a subsystem or model
scalar

Control signal (trigger or function-call) attached externally to the outside of an Subsystem block or a Model block that is passed to the inside of the block. To enable this port, select Show output port.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | expression
Parameters

Main

**Trigger type — Select the type control signal**

*rising* (default) | *falling* | *either* | *function-call*

Select the type of control signal that executes a subsystem or model.

**rising**
Triggers execution of subsystem or model when the control signal rises from a negative or zero value to a positive value. If the initial value is negative, a rising signal to zero triggers execution.

**falling**
Triggers execution of subsystem or model when the control signal falls from a positive or a zero value to a negative value. If the initial value is positive, a falling signal to zero triggers execution.

**either**
Triggers execution of subsystem or model when the control signal is either rising or falling.

**function-call**
Execute subsystem or model when the control signal receives a function-call event from a Stateflow chart, Function-Call Generator block, or an S-function block.

**Programmatic Use**

**Block Parameter:** TriggerType  
**Type:** character vector  
**Values:** 'rising' | 'falling' | 'either' | 'function-call'  
**Default:** 'rising'

**Treat as Simulink function — Create Simulink Function block**

*off* (default) | *on*

Create a Simulink Function block by configuring a Subsystem block that is callable with arguments from a function caller:

- **off**
  Remove configuration.
Configure a Subsystem block as a Simulink Function block. The Trigger block must reside within the subsystem.

You can edit the function prototype that displays on the block face to specify input and output arguments for the block.

**Dependency**

To display and enable this parameter, select function-call from the **Trigger type** list.

**Programmatic Use**

**Block Parameter:** IsSimulinkFunction  
**Type:** character vector  
**Values:** 'off' | 'on'  
**Default:** 'off'

**Function name — Specify function name for Simulink Function block**  
**f** (default) | function name

Specify the function name for a Simulink Function block. Alternatively, you can specify the name by editing the function prototype on the face of the block.

**f**

Default name for a Simulink Function block.

**function name**

Function name that displays on the face of a Simulink Function block.

**Dependency**

To display and enable this parameter, select function-call from the **Trigger type** list and select the **Treat as a Simulink Function** check box.

**Programmatic Use**

**Block Parameter:** FunctionName  
**Type:** character vector  
**Values:** 'f' | '<function name>'  
**Default:** 'f'

**Enable variant condition — Controls activating the variant control (condition)**  
on (default) | off
Control activating the variant control (condition) defined with the **Variant Control** parameter.

- **off**
  Deactivate variant control of subsystem.

- **on**
  Activate variant control of subsystem. Selecting this parameter:
  - Enables the **Variant control** parameter.
  - Displays a variant badge on the face of the block indicating variant conditions are enabled.

**Dependency**

To display and enable this parameter, select function-call from the **Trigger type** list and select the **Treat as a Simulink Function** check box.

**Programmatic Use**

**Block Parameter:** Variant  
**Type:** character vector  
**Values:** 'off' | 'on'  
**Default:** 'off'

**Variant control — Specify variant control (condition) expression**

**(inherit)** (default) | <logical expression>

Specify variant control (condition) expression that executes a variant Simulink Function block when the expression evaluates to true.

**(inherit)**

Default value for variant control. Inherits the variant condition from the corresponding Function Caller blocks in the model. When **Variant Control** is set as **(inherit)** the value for **Generate preprocessor conditionals** is inherited automatically from the Function Caller block in the model.

**logical expression**

A logical (Boolean) expression or a Simulink.Variant object representing a logical expression.

The function is activated when the expression evaluates to true.
If you want to generate code for your model, define the variables in the expression as Simulink.Parameter objects.

**Dependency**

To display and enable this parameter, select function-call from the Trigger type list, select the Treat as a Simulink Function check box and then select the Enable variant condition check box.

**Programmatic Use**

**Block Parameter:** VariantControl  
**Type:** character vector  
**Values:** '(inherit)|<logical expression>|Simulink.Variant object  
**Default:** '(inherit)'

**Generate preprocessor conditionals — Control enclosing variant choices**

off (default) | on

Control enclosing variant choices within C preprocessor conditional statements.

- **off**
  - Do not enclose variant choices within C preprocessor conditional statements.

- **on**
  - When generating code for an ERT target, enclose variant choices within C preprocessor conditional statements (#if).

**Dependency**

To display and enable this parameter, select the Enable variant condition check box.

**Programmatic Use**

**Block Parameter:** GeneratePreprocessorConditionals  
**Type:** character vector  
**Values:** 'off' | 'on'  
**Default:** 'off'

**Function visibility — Select scope visibility of function**

scoped (default) | global

Select scope of Simulink Function block within subsystem or model.
scoped
  Limit accessibility of function to:
  • Hierarchic level containing the Simulink Function block and levels below.
  • One hierarchical level above with qualification.

global
  Function accessible from any part of the model hierarchy.

Dependency
To display and enable this parameter, select function-call from the Trigger type list and then select the Treat as a Simulink Function check box..

Programmatic Use
Block Parameter: FunctionVisibility
  Type: character vector
  Values: 'scoped' | 'global'
  Default: 'scoped'

States when enabling — Select how to set block state values
held (default) | reset | inherit
Select how to set block state values when the subsystem or model is disabled.

held
  Leave the block states at their current values.
reset
  Reset the block state values.
inherit
  Use the held or reset setting from the parent subsystem initiating the function-call. If the parent of the initiator is the model root, the inherited setting is held. If the trigger has multiple initiators, set the parents of all initiators to either held or reset.

Dependencies
To enable this parameter, select function-call from the Trigger Type list.
This parameter setting applies only if the model explicitly enables and disables the function-call subsystem. For example:
• The function-call subsystem resides in an enabled subsystem. In this case, the model enables and disables the function-call subsystem along with the parent subsystem.

• The function-call initiator that controls the function-call subsystem resides in an enabled subsystem. In this case, the model enables and disables the function-call subsystem along with the enabled subsystem containing the function-call initiator.

• The function-call initiator is a Stateflow event bound to a particular state. See “Control Function-Call Subsystems by Using bind Actions” (Stateflow).

• The function-call initiator is an S-function that explicitly enables and disables the function-call subsystem. See `ssEnableSystemWithTid` for an example.

**Programmatic Use**

**Block Parameter:** StatesWhenEnabling

**Type:** character vector

**Values:** 'held' | 'reset' | 'inherit'

**Default:** 'held'

**Propagate sizes of variable-size signals — Select when to propagate variable-size signals**

**During execution** (default) | **Only when enabling**

Select when to propagate variable-size signals.

**During execution**

Propagate variable-size signals at each time step.

**Only when enabling**

Propagate variable-size signals when executing a Subsystem block or Model block containing an Enable port, Trigger port with Trigger type set to function-call, or Action Port block. When you select this option, sample time must be periodic.

**Dependencies**

To display and enable this parameter for a Trigger port block, select Function-call from the Trigger type list.

**Programmatic Use**

**Block Parameter:** PropagateVarSize

**Type:** character vector

**Values:** 'During execution' | 'Only when enabling'

**Default:** 'During execution'
Show output port — Control display of output port
off (default) | on

Control display of an output port for a signal that identifies the trigger signal or function-call event signal.

- off
  Remove the output port.

- on
  Display the output port and determine which signal caused the trigger signal or function-call event. The width of the output port signal is the width of the control signal. The signal value is:
  
  - 1 for a signal that causes a rising trigger
  - -1 for a signal that causes a falling trigger
  - 2 for a function-call event
  - 0 in all other cases

Programmatic Use

Block Parameter: ShowOutputPort
Type: character vector
Values: 'off' | 'on'
Default: 'off'

Output data type — Select output port data type
auto (default) | double | int8

Select output port data type for the signal that identifies the trigger signal or function-call event signal.

- auto
  Data type is the same as the port connected to the output.

- double
  Double value.

- int8
  Integer value
**Dependency**

To enable this parameter, select the **Show output port** check box.

The Trigger block ignores the **Data type override** setting for the Fixed-Point Tool.

**Programmatic Use**

**Block Parameter**: OutputDataType

**Type**: character vector

**Values**: 'auto' | 'double' | 'int8'

**Default**: 'auto'

**Sample time type — Select calling rate**

triggered (default) | periodic

Select the calling rate for a subsystem or model.

**triggered**

Apply to applications that do not have a periodic function-call frequency. A function-call initiator can execute a triggered (aperiodic) function-call subsystem one or more times per time step and can provide a series of aperiodic function-call events.

**periodic**

A function-call initiator can execute a periodic function-call system only once per time step and must provide a series of periodic function-call events. A Stateflow chart is an example of a function-call initiator.

**Dependency**

To enable this parameter, select **Function-call** from the **Trigger type** list.

**Programmatic Use**

**Block Parameter**: SampleTimeType

**Type**: character vector

**Values**: 'triggered' | 'periodic'

**Default**: 'triggered'

**Sample time — Specify time interval**

-1 (default) | Ts | [Ts, To]

Specify the time interval between function calls to a subsystem or model containing this Trigger port block. If the actual calling rate for the subsystem or model differs from the time interval this parameter specifies, Simulink displays an error.
Inherit time interval from the control signal.

**Ts**
Scalar where Ts is the time interval.

**[Ts, To]**
Vector where Ts is the time interval and To is the initial time offset.

**Dependencies**
To enable this parameter, select function-call from the Trigger type list and periodic from the Sample time type list.

**Programmatic Use**
**Block Parameter:** SampleTime  
**Type:** character vector  
**Values:** '-1' | 'Ts' | '[Ts, To]'  
**Default:** '-1'

**Enable zero-crossing detection — Control zero-crossing detection**

on (default) | off

Control .

on
Detect zero crossings.

off
Do not detect zero crossings.

**Dependencies**
To enable this parameter, select rising, falling, or either from the Trigger type list.

**Programmatic Use**
**Block Parameter:** ZeroCross  
**Type:** character vector  
**Values:** 'on' | 'off'  
**Default:** 'on'
Initial trigger signal state — Select the initial state of the trigger signal
compatibility (no trigger on first evaluation) (default) | zero | positive | negative

Select the initial state of the trigger control signal.

compatibility (no trigger on first evaluation)
   No trigger at the first evaluation of trigger signal. If you choose this option and the
   Trigger block is in a subsystem where the states are reset, the block does not reset.
zero
   Zero. Helps to evaluate a rising or falling trigger signal at the first time step.
positive
   Positive value. Helps to evaluate a falling trigger signal at the first time step.
negative
   Negative value. Helps to evaluate a rising trigger signal at the first time step.

Dependency

To display and activate this parameter, select rising, falling, or either from the
Trigger type list.

Programmatic Use
Block Parameter: InitialTriggerSignalState
Type: character vector
Values: 'compatibility (no trigger on first evaluation)' | 'zero' | 'positive' | 'negative'
Default: 'compatibility (no trigger on first evaluation)'

Signal Attribute

Port dimensions — Specify dimensions for the trigger signal
1 (default) | [n] | [m n]

Specify dimensions for the trigger signal attached externally a Model block and passed to
the inside of the block.

1
   Scalar signal.
[n]
   Vector signal of width n.

[m n]
   Matrix signal having m rows and n columns.

**Dependency**

To display and enable this parameter for a Trigger port block at the root-level of a model, select rising, falling, or either from the **Trigger type** list.

**Programmatic Use**

*Block Parameter*: PortDimensions

*Type*: character vector

*Values*: '1' | '[n]' | '[m n]'  

*Default*: '1'

**Trigger signal sample time — Specify time interval**

-1 (default) | Ts | [Ts, To]

Specify time interval between block method executions for the block driving the trigger signal.

-1
   Inherit time interval.

Ts
   Scalar where Ts is the time interval.

[Ts, To]
   Vector where Ts is the time interval and To is the initial time offset.

**Dependency**

To display and enable this parameter for a Trigger port block at the root-level of a model, select rising, falling, or either from the **Trigger type** list.

**Programmatic Use**

*Block Parameter*: TriggerSignalSampleTime

*Type*: character vector

*Values*: '-1' | 'Ts' | '[Ts, To]'  

*Default*: '-1'
**Minimum — Specify minimum output value for the trigger signal**

[] (default) | real scalar

Specify minimum value for the trigger signal attached externally to a Model block and passed to the inside of the block.

Simulink uses this value to perform:

- Simulation range checking. See “Signal Ranges”.
- Automatic scaling of fixed-point data types.
- Optimization of generated code. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. See “Optimize using the specified minimum and maximum values” (Simulink Coder).

[]

Unspecified minimum value.
real scalar
  Real scalar value.

**Dependency**

To display and enable this parameter for a Trigger port block at the root-level of a model, select rising, falling, or either from the **Trigger type** list.

**Programmatic Use**

**Block Parameter:** OutMin  
**Type:** character vector  
**Values:** '[]' | '<real scalar>'  
**Default:** '[]'

**Maximum — Specify maximum output value for the trigger signal**

[] (default) | real scalar

Specify maximum value for the trigger signal attached externally to a Model block and passed to the inside of the block.

Simulink uses this value to perform:

- Simulation range checking. See “Signal Ranges”.
- Automatic scaling of fixed-point data types.
• Optimization of generated code. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. See “Optimize using the specified minimum and maximum values” (Simulink Coder).

[]

Unspecified maximum value.
real scalar
  Real scalar value.

**Dependency**

To display and enable this parameter for a Trigger port block at the root-level of a model, select rising, falling, or either from the **Trigger type** list.

**Programmatic Use**

**Block Parameter:** OutMax  
**Type:** character vector  
**Values:** '[]' | '<real scalar>'  
**Default:** '[]'

**Data type — Select output data type for the trigger signal**

double (default) | single | int8 | uint8 | int16 | uint16 | int32 | uint32 | int64 | uint64 | boolean | fixdt(1,16) | fixdt(1,16,0) | fixdt(1,16,2^,0) | <data type expression>

Select data type for the trigger signal attached externally to a Model block and passed to the inside of the block.

double
  Double-precision floating point.
single
  Single-precision floating point.
int8
  Signed 8-bit integer.
uint8
  Unsigned 8-bit integer.
int16
  Signed 16-bit integer.
uint16
  Unsigned 16-bit integer.
int32
  Signed 32-bit integer.
uint32
  Unsigned 32-bit integer.
int64
  Signed 64-bit integer.
uint64
  Unsigned 64-bit integer.
boolean
  Boolean with a value of true or false.
fixdt(1,16)
  Signed 16-bit fixed point number with binary point undefined.
fixdt(1,16,0)
  Signed 16-bit fixed point number with binary point set to zero.
fixdt(1,16,2^0,0)
  Signed 16-bit fixed point number with slope set to 2^0 and bias set to 0.
<data type expression>
  Data type object, for example Simulink.NumericType. Do not specify a bus object as the expression.

Dependency
To display and enable this parameter for a Trigger port block at the root-level of a model, select rising, falling, or either from the Trigger type list.

Programmatic Use
Block Parameter: OutDataTypeStr
Type: character vector
Values: 'double' | 'single' | 'int8' | 'uint8' | 'int16' | 'uint16' | 'int32' | 'uint32' | 'int64' | 'uint64' | 'boolean' | 'fixdt(1,16)' | 'fixdt(1,16,0)' | 'fixdt(1,16,2^0,0)' | '<data type expression>'
Default: 'double'
**Mode — Select data type category**  
**Build in (default) | Fixed point | Expression**

Select data type category and display drop-down lists to help you define the data type.

- **Build in**
  Display drop-down lists for data type and **Data type override**.
- **Fixed point**
  Display drop-down lists for **Signedness**, **Scaling**, and **Data type override**.
- **Expression**
  Display text box for entering an expression.

**Dependency**

To enable this parameter, select the Show data type assistant button.

**Programmatic Use**

No equivalent command-line parameter.

**Interpolate data — Control how missing workspace data is estimated**  
**on (default) | off**

Control how missing workspace data is estimated when loading data from the MATLAB workspace.

- **on**
  Linearly interpolate output at time steps for which no corresponding workspace data exists.
- **off**
  Do not interpolate output at time steps. The current output equals the output at the most recent time step for which data exists.

**Dependency**

To display and enable this parameter for a Trigger port block at the root-level of a model, select **rising**, **falling**, or **either** from the **Trigger type** list.
Programmatic Use

**Block Parameter**: Interpolate
**Type**: character vector
**Values**: 'on' | 'off'
**Default**: 'on'

Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>double</th>
<th>enumerated</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
</table>

| Direct Feedthrough  | no      |
| Multidimensional Signals | yes    |
| Variable-Size Signals  | no      |
| Zero-Crossing Detection | yes    |

Extended Capabilities

**C/C++ Code Generation**
Generate C and C++ code using Simulink® Coder™.

**HDL Code Generation**
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic. See also “HDL Code Generation” on page 1-2196.
HDL Architecture

This block has a single, default HDL architecture.

PLC Code Generation

generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

Design and simulate fixed-point systems using Fixed-Point Designer™.

See Also

Enabled and Triggered Subsystem | Function-Call Subsystem | Subsystem | Triggered Subsystem

Topics

“Conditionally Executed Subsystems Overview”
“Using Triggered Subsystems”
“Using Enabled and Triggered Subsystems”
“Using Function-Call Subsystems”
“Export-Function Models Overview”

Introduced before R2006a
Trigger-Based Linearization

Generate linear models in base workspace when triggered
Library: Simulink / Model-Wide Utilities

Description

When triggered, this block calls linmod or dlinmod to create a linear model for the system at the current operating point. No trimming is performed. The linear model is stored in the base workspace as a structure, along with information about the operating point at which the snapshot was taken. Multiple snapshots are appended to form an array of structures.

The block sets the following model parameters to the indicated values:

- BufferReuse = 'off'
- RTWInlineParameters = 'on'
- BlockReductionOpt = 'off'

The name of the structure used to save the snapshots is the name of the model appended by _Trigger_Based_Linearization, for example, vdp_Trigger_Based_Linearization. The structure has the following fields:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>The A matrix of the linearization</td>
</tr>
<tr>
<td>b</td>
<td>The B matrix of the linearization</td>
</tr>
<tr>
<td>c</td>
<td>The C matrix of the linearization</td>
</tr>
<tr>
<td>d</td>
<td>The D matrix of the linearization</td>
</tr>
<tr>
<td>StateName</td>
<td>Names of the model's states</td>
</tr>
<tr>
<td>Field</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>OutputName</td>
<td>Names of the model's output ports</td>
</tr>
<tr>
<td>InputName</td>
<td>Names of the model's input ports</td>
</tr>
<tr>
<td>OperPoint</td>
<td>A structure that specifies the operating point of the linearization. The structure specifies the value of the model's states (OperPoint.x) and inputs (OperPoint.u) at the operating point time (OperPoint.t).</td>
</tr>
<tr>
<td>Ts</td>
<td>The sample time of the linearization for a discrete linearization</td>
</tr>
</tbody>
</table>

**Tip** Use the Timed-Based Linearization block to generate linear models at predetermined times.

**Ports**

**Input**

**Port_1 — Input signal**

 scalar  

Input trigger signal specified as a scalar. You specify the type of event that triggers generation of a linear model using the **Trigger type** parameter.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | fixed point

**Parameters**

**Trigger type — Signal type that triggers generation of a linear model**

 rising | falling | either | function-call  

Type of event on the trigger input signal that triggers generation of a linear model. You can select:

- **rising** — Trigger execution of subsystem or model when the trigger signal rises from a negative or zero value to a positive value. If the initial value is negative, a rising signal to zero triggers execution.
- **falling** — Trigger execution of subsystem or model when the trigger signal falls from a positive or a zero value to a negative value. If the initial value is positive, a falling signal to zero triggers execution.

- **either** — Trigger execution of subsystem or model when the trigger signal is either rising or falling.

- **function-call** — Trigger execution of subsystem or model when the trigger signal is a function-call event from a Stateflow chart, Function-Call Generator block, or an S-Function.

**Programmatic Use**

**Block Parameter:** TriggerType  
**Type:** character vector  
**Values:** 'rising' | 'falling' | 'either' | 'function-call'  
**Default:** 'rising'

**Sample time (of linearized model) — Sample time**

0 (default) | scalar | vector

Specify a sample time for the linear model. To create a continuous-time model using linmod, specify a sample time of 0. Otherwise, to create a discrete-time model using dlinmod, specify a positive sample time (see “Discrete-Time System Linearization” on page 2-58).

**Programmatic Use**

**Block Parameter:** SampleTime  
**Type:** character vector  
**Values:** scalar | vector  
**Default:** '0'

**Block Characteristics**

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>double</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Extended Capabilities

Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

See Also
Timed-Based Linearization | dlinmod | linmod

Topics
“Discrete-Time System Linearization” on page 2-58

Introduced before R2006a
Triggered Subsystem

Subsystem whose execution is triggered by external input

**Library:**
- Simulink / Ports & Subsystems
- HDL Coder / Ports & Subsystems

**Description**

The Triggered Subsystem block is a Subsystem block preconfigured as a starting point for creating a subsystem that executes each time the control signal has a trigger value.

Use Trigger Subsystem blocks to model:

- A task that runs with the detection of a trigger value.
- An interrupt from I/O hardware.
- A processor request to handle an exception or error.
Ports

Input

**In — Signal input to a subsystem block**

scalar | vector | matrix

Placing an Inport block in a subsystem block adds an external input port to the block. The port label matches the name of the Inport block.

Use Inport blocks to get signals from the local environment.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus

**Trigger — Control signal input to a subsystem block**

scalar

Placing a Trigger block in a subsystem block adds an external input port to the block.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

Output

**Out — Signal output from a subsystem**

scalar | vector | matrix

Placing an Outport block in a subsystem block adds an output port from the block. The port label on the subsystem block is the name of the Outport block.

Use Outport blocks to send signals to the local environment.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus
Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>bus</th>
<th>double</th>
<th>enumerated</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
<th>string</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>limited</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>limited</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

a. Actual data type or capability support depends on block implementation.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

Actual code generation support depends on block implementation.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic. For information about best practices, restrictions, and how you can use the trigger signal as a clock with the TriggerAsClock property, see “Using Triggered Subsystems for HDL Code Generation” (HDL Coder).
### HDL Architecture

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Module (default)</td>
<td>Generate code for the subsystem and the blocks within the subsystem.</td>
</tr>
<tr>
<td>BlackBox</td>
<td>Generate a black box interface. The generated HDL code includes only the input/output port definitions for the subsystem. Therefore, you can use a subsystem in your model to generate an interface to existing, manually written HDL code. The black-box interface generation for subsystems is similar to the Model block interface generation without the clock signals.</td>
</tr>
<tr>
<td>No HDL</td>
<td>Remove the subsystem from the generated code. You can use the subsystem in simulation, however, treat it as a “no-op” in the HDL code.</td>
</tr>
</tbody>
</table>

### Black Box Interface Customization

For the BlackBox architecture, you can customize port names and set attributes of the external component interface. See “Customize Black Box or HDL Cosimulation Interface” (HDL Coder).

### HDL Block Properties

<table>
<thead>
<tr>
<th>General</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>AdaptivePipelining</td>
<td>Automatic pipeline insertion based on the synthesis tool, target frequency, and multiplier word-lengths. The default is inherit. See also “AdaptivePipelining” (HDL Coder).</td>
</tr>
<tr>
<td>BalanceDelays</td>
<td>Detects introduction of new delays along one path and inserts matching delays on the other paths. The default is inherit. See also “BalanceDelays” (HDL Coder).</td>
</tr>
<tr>
<td>ClockRatePipelining</td>
<td>Insert pipeline registers at a faster clock rate instead of the slower data rate. The default is inherit. See also “ClockRatePipelining” (HDL Coder).</td>
</tr>
<tr>
<td>ConstrainedOutputPipeline</td>
<td>Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. For more details, see “ConstrainedOutputPipeline” (HDL Coder).</td>
</tr>
</tbody>
</table>
### General

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>DistributedPipelining</strong></td>
<td>Pipeline register distribution, or register retiming. The default is off. See also “DistributedPipelining” (HDL Coder).</td>
</tr>
<tr>
<td><strong>DSPStyle</strong></td>
<td>Synthesis attributes for multiplier mapping. The default is none. See also “DSPStyle” (HDL Coder).</td>
</tr>
<tr>
<td><strong>FlattenHierarchy</strong></td>
<td>Remove subsystem hierarchy from generated HDL code. The default is inherit. See also “FlattenHierarchy” (HDL Coder).</td>
</tr>
<tr>
<td><strong>InputPipeline</strong></td>
<td>Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “InputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td><strong>OutputPipeline</strong></td>
<td>Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “OutputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td><strong>SharingFactor</strong></td>
<td>Number of functionally equivalent resources to map to a single shared resource. The default is 0. See also “Resource Sharing” (HDL Coder).</td>
</tr>
<tr>
<td><strong>StreamingFactor</strong></td>
<td>Number of parallel data paths, or vectors, that are time multiplexed to transform into serial, scalar data paths. The default is 0, which implements fully parallel data paths. See also “Streaming” (HDL Coder).</td>
</tr>
</tbody>
</table>

### Target Specification

This block cannot be the DUT, so the block property settings in the **Target Specification** tab are ignored.

### Fixed-Point Conversion

Design and simulate fixed-point systems using Fixed-Point Designer™.

Actual data type support depends on block implementation.
See Also

**Blocks**
Enabled Subsystem | Enabled and Triggered Subsystem | Function-Call Subsystem | Subsystem | Trigger

**Topics**
“Conditionally Executed Subsystems Overview”
“Using Enabled Subsystems”
“Using Triggered Subsystems”
“Using Enabled and Triggered Subsystems”
“Using Function-Call Subsystems”

**Introduced before R2006a**
Trigonometric Function

Specified trigonometric function on input

**Library:**
- Simulink / Math Operations
- HDL Coder / Math Operations

## Description

The Trigonometric Function block performs common trigonometric functions and outputs the result in rad.

## Supported Functions

You can select one of these functions from the **Function** parameter list.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Mathematical Expression</th>
<th>MATLAB Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>sin</td>
<td>Sine of the input</td>
<td>sin(u)</td>
<td>sin</td>
</tr>
<tr>
<td>cos</td>
<td>Cosine of the input</td>
<td>cos(u)</td>
<td>cos</td>
</tr>
<tr>
<td>tan</td>
<td>Tangent of the input</td>
<td>tan(u)</td>
<td>tan</td>
</tr>
<tr>
<td>asin</td>
<td>Inverse sine of the input</td>
<td>asin(u)</td>
<td>asin</td>
</tr>
<tr>
<td>acos</td>
<td>Inverse cosine of the input</td>
<td>acos(u)</td>
<td>acos</td>
</tr>
<tr>
<td>atan</td>
<td>Inverse tangent of the input</td>
<td>atan(u)</td>
<td>atan</td>
</tr>
<tr>
<td>atan2</td>
<td>Four-quadrant inverse tangent of the input</td>
<td>atan2(u)</td>
<td>atan2</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
<td>Mathematical Expression</td>
<td>MATLAB Equivalent</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
<td>-------------------------</td>
<td>-------------------</td>
</tr>
<tr>
<td>sinh</td>
<td>Hyperbolic sine of the input</td>
<td>sinh(u)</td>
<td>sinh</td>
</tr>
<tr>
<td>cosh</td>
<td>Hyperbolic cosine of the input</td>
<td>cosh(u)</td>
<td>cosh</td>
</tr>
<tr>
<td>tanh</td>
<td>Hyperbolic tangent of the input</td>
<td>tanh(u)</td>
<td>tanh</td>
</tr>
<tr>
<td>asinh</td>
<td>Inverse hyperbolic sine of the input</td>
<td>asinh(u)</td>
<td>asinh</td>
</tr>
<tr>
<td>acosh</td>
<td>Inverse hyperbolic cosine of the input</td>
<td>acosh(u)</td>
<td>acosh</td>
</tr>
<tr>
<td>atanh</td>
<td>Inverse hyperbolic tangent of the input</td>
<td>atanh(u)</td>
<td>atanh</td>
</tr>
<tr>
<td>sincos</td>
<td>Sine of the input; cosine of the input</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>cos + jsin</td>
<td>Complex exponential of the input</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>

**CORDIC Approximation Method**

If you use the CORDIC approximation method (see “More About” on page 1-2211), the block input has some further requirements.

When you set **Function** to sin, cos, sincos, or cos + jsin, and set the **Approximation method** to CORDIC, the block has these limitations:

- When you use signed fixed-point types, the input angle must fall within the range \([-2\pi, 2\pi)\) rad.
- When you use unsigned fixed-point types, the input angle must fall within the range \([0, 2\pi)\) rad.

When you set **Function** to atan2 and the **Approximation method** to CORDIC, the block has these limitations:

- Inputs must be the same size, or at least one value must be a scalar value.
• Both inputs must have the same data type.
• When you use signed fixed-point types, the word length must be 126 or less.
• When you use unsigned fixed-point types, the word length must be 125 or less.

This table summarizes what happens for an invalid input.

<table>
<thead>
<tr>
<th>Block Usage</th>
<th>Effect of Invalid Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simulation</td>
<td>An error appears.</td>
</tr>
<tr>
<td>Generated code</td>
<td>Undefined behavior occurs. Avoid relying on undefined behavior for generated code or accelerator modes.</td>
</tr>
<tr>
<td>Accelerator modes</td>
<td></td>
</tr>
</tbody>
</table>

## Ports

### Input

**Port_1 — Input signal**

scalar | vector | matrix

Input specified as a scalar, vector, or matrix. The block accepts input signals of the following data types:

<table>
<thead>
<tr>
<th>Functions</th>
<th>Input Data Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>• sin</td>
<td>• Floating point</td>
</tr>
<tr>
<td>• cos</td>
<td>• Fixed point (only when Approximation method is CORDIC)</td>
</tr>
<tr>
<td>• sincos</td>
<td></td>
</tr>
<tr>
<td>• cos + jsin</td>
<td></td>
</tr>
<tr>
<td>• atan2</td>
<td></td>
</tr>
</tbody>
</table>
**Functions**

- tan
- asin
- acos
- atan
- sinh
- cosh
- tanh
- asinh
- acosh
- atanh

**Input Data Types**

- Floating point

**Dependencies**

- When you set Function to atan2, the block shows two input ports. The first input (Port_1) is the y-axis or imaginary part of the function argument. The second input (Port_2) is the x-axis or real part of the function argument.
- You can use floating-point input signals when you set Approximation method to None or CORDIC. However, the block output data type depends on which of these approximation method options you choose.

<table>
<thead>
<tr>
<th>Input Data Type</th>
<th>Approximation Method</th>
<th>Output Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Floating point</td>
<td>None</td>
<td>Depends on your selection for Output signal type. Options are auto (same data type as input), real, or complex.</td>
</tr>
<tr>
<td>Floating point</td>
<td>CORDIC</td>
<td>Same as input. Output signal type is not available when you use the CORDIC approximation method to compute the block output.</td>
</tr>
</tbody>
</table>

For CORDIC approximations:

- Input must be real for the sin, cos, sincos, cos + jsin, and atan2 functions.
• Output is real for the \( \sin, \cos, \text{sincos}, \) and \( \text{atan2} \) functions.
• Output is complex for the \( \cos + j\sin \) function.

**Limitations**

Complex input signals are supported for all functions in this block, except \( \text{atan2} \).

You can use fixed-point input signals only when **Approximation method** is set to **CORDIC**. The CORDIC approximation is available for the \( \sin, \cos, \text{sincos}, \cos + j\sin, \) and \( \text{atan2} \) functions. For the \( \text{atan2} \) function, the relationship between input and output data types depends also on whether the fixed-point input is signed or unsigned.

<table>
<thead>
<tr>
<th>Input Data Type</th>
<th>Function</th>
<th>Output Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fixed point, signed or unsigned</td>
<td>( \sin, \cos, \text{sincos}, ) and ( \cos + j\sin )</td>
<td>\text{fixdt}(1, WL, WL - 2) where WL is the input word length. This fixed-point type provides the best precision for the CORDIC algorithm.</td>
</tr>
<tr>
<td>Fixed point, signed</td>
<td>( \text{atan2} )</td>
<td>\text{fixdt}(1, WL, WL - 3)</td>
</tr>
<tr>
<td>Fixed point, unsigned</td>
<td>( \text{atan2} )</td>
<td>\text{fixdt}(1, WL, WL - 2)</td>
</tr>
</tbody>
</table>

When you set **Function** to \( \sin, \cos, \text{sincos}, \) or \( \cos + j\sin \), and set the **Approximation method** to **CORDIC**, the block has these limitations:

• When you use signed fixed-point types, the input angle must fall within the range \([ -2\pi, 2\pi) \) rad.
• When you use unsigned fixed-point types, the input angle must fall within the range \([ 0, 2\pi) \) rad.

When you set **Function** to \( \text{atan2} \) and the **Approximation method** to **CORDIC**, the block has these limitations:

• Inputs must be the same size, or at least one value must be a scalar value.
• Both inputs must have the same data type.
• When you use signed fixed-point types, the word length must be 126 or less.
• When you use unsigned fixed-point types, the word length must be 125 or less.

**Data Types:** single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point
Port_2 — x-axis or real part of the function argument for atan2
scalar | vector | matrix

Input the x-axis or real part of the function argument for atan2. When you set Function to atan2, the block shows two input ports. The first input (Port_1) is the y-axis or imaginary part of the function argument. The second input (Port_2) is the x-axis or real part of the function argument. (See “Port Location After Rotating or Flipping” for a description of the port order for various block orientations.)

Dependencies

To enable this port, set Function to atan2.

Limitations

- Fixed-point input signals are supported only when you set Approximation method to CORDIC.
- When you set Function to atan2 and Approximation method to CORDIC:
  - Inputs must be the same size, or at least one value must be a scalar value.
  - Both inputs must have the same data type.
  - When you use signed fixed-point types, the word length must be 126 or less.
  - When you use unsigned fixed-point types, the word length must be 125 or less.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

Output

Port_1 — Specified trigonometric function of input
scalar | vector | matrix

Result of applying the specified trigonometric function to one or more inputs in rad. Each function supports:

- Scalar operations
- Element-wise vector and matrix operations

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point
**sin — Sine of input signal**  
scalar | vector | matrix

Sine of the input signal, in rad.

**Dependencies**

To enable this port, set **Function** to sincos.

**Limitations**

Fixed-point input signals are supported only when you set **Approximation method** to CORDIC.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

**cos — Cosine of input signal**  
scalar | vector | matrix

Cosine of the input signal, in rad.

**Dependencies**

To enable this port, set **Function** to sincos.

**Limitations**

Fixed-point input signals are supported only when you set **Approximation method** to CORDIC.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

**Parameters**

**Function — Trigonometric function**

sin (default) | cos | tan | asin | acos | atan | atan2 | sinh | cosh | tanh | asinh | acosh | atanh | sincos | cos + jsin

Specify the trigonometric function. The name of the function on the block icon changes to match your selection.
Limitations

When you set **Function** to sin, cos, sincos, or cos + jsin, and set the **Approximation method** to CORDIC, the block has these limitations:

• When you use signed fixed-point types, the input angle must fall within the range \([-2\pi, 2\pi)\) rad.
• When you use unsigned fixed-point types, the input angle must fall within the range \([0, 2\pi)\) rad.

When you set **Function** to atan2 and the **Approximation method** to CORDIC, the block has these limitations:

• Inputs must be the same size, or at least one value must be a scalar value.
• Both inputs must have the same data type.
• When you use signed fixed-point types, the word length must be 126 or less.
• When you use unsigned fixed-point types, the word length must be 125 or less.

Programmatic Use

**Block Parameter:** Operator

**Type:** character vector

**Values:** 'sin' | 'cos' | 'tan' | 'asin' | 'acos' | 'atan' | 'atan2' | 'sinh' | 'cosh' | 'tanh' | 'asinh' | 'acosh' | 'atanh' | 'sincos' | 'cos + jsin'

**Default:** 'sin'

**Approximation method — CORDIC or none**

None (default) | CORDIC

Specify the type of approximation for computing output.

<table>
<thead>
<tr>
<th>Approximation Method</th>
<th>Data Types Supported</th>
<th>When to Use This Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>None (default)</td>
<td>Floating point</td>
<td>You want to use the default Taylor series algorithm.</td>
</tr>
<tr>
<td>CORDIC</td>
<td>Floating point and fixed point</td>
<td>You want a fast, approximate calculation.</td>
</tr>
</tbody>
</table>

If you select CORDIC and enlarge the block from the default size, the block icon changes:
<table>
<thead>
<tr>
<th>Function</th>
<th>Block Icon</th>
</tr>
</thead>
<tbody>
<tr>
<td>sin</td>
<td><img src="image" alt="Trigonometric Function" /></td>
</tr>
<tr>
<td>cos</td>
<td><img src="image" alt="Trigonometric Function" /></td>
</tr>
<tr>
<td>sincos</td>
<td><img src="image" alt="Trigonometric Function" /></td>
</tr>
<tr>
<td>cos + jsin</td>
<td><img src="image" alt="Trigonometric Function" /></td>
</tr>
<tr>
<td>atan2</td>
<td><img src="image" alt="Trigonometric Function" /></td>
</tr>
</tbody>
</table>

**Dependencies**

To enable this parameter, set **Function** to **sin**, **cos**, **sincos**, **cos + jsin**, or **atan2**.

To use fixed-point input signals, you must set **Approximation method** to CORDIC.
**Limitations**

When you set **Function** to \( \sin, \cos, \text{sincos}, \text{or cos} + j\sin \), and set the **Approximation method** to CORDIC, the block has these limitations:

- When you use signed fixed-point types, the input angle must fall within the range \([-2\pi, 2\pi)\) rad.
- When you use unsigned fixed-point types, the input angle must fall within the range \([0, 2\pi)\) rad.

When you set **Function** to \( \text{atan2} \) and the **Approximation method** to CORDIC, the block has these limitations:

- Inputs must be the same size, or at least one value must be a scalar value.
- Both inputs must have the same data type.
- When you use signed fixed-point types, the word length must be 126 or less.
- When you use unsigned fixed-point types, the word length must be 125 or less.

**Programmatic Use**

**Block Parameter:** ApproximationMethod

Type: character vector

**Values:** 'None' | 'CORDIC'

**Default:** 'None'

**Number of iterations** — Number of iterations for CORDIC algorithm

11 (default) | positive integer, less than or equal to word length of fixed-point input

Specify the number of iterations to perform the CORDIC algorithm. The default value is 11.

- When the block input uses a floating-point data type, the number of iterations can be a positive integer.
- When the block input is a fixed-point data type, the number of iterations cannot exceed the word length.

For example, if the block input is \( \text{fixdt}(1,16,15) \), the word length is 16. In this case, the number of iterations cannot exceed 16.

**Dependencies**

To enable this parameter, you must set the **Function** and **Approximation method** parameters as follows:
• Set **Function** to sin, cos, sincos, cos + jsin, or atan2.
• Set **Approximation method** to CORDIC.

**Programmatic Use**
**Block Parameter:** NumberOfIterations  
**Type:** character vector  
**Values:** positive integer, less than or equal to word length of fixed-point input  
**Default:** '11'

**Output signal type — complexity of output signal**
*auto* (default) | *real* | *complex*

Specify the output signal type of the Trigonometric Function block as *auto*, *real*, or *complex*.

<table>
<thead>
<tr>
<th>Function</th>
<th>Input Signal Type</th>
<th>Output Signal Type</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Auto</td>
</tr>
<tr>
<td>Any selection for the <strong>Function</strong> parameter</td>
<td>real</td>
<td>real</td>
</tr>
<tr>
<td></td>
<td>complex</td>
<td>complex</td>
</tr>
</tbody>
</table>

**Dependencies**

Setting **Approximation method** to CORDIC disables this parameter.

**Note** When **Function** is atan2, complex input signals are not supported for simulation or code generation.

**Programmatic Use**
**Block Parameter:** OutputSignalType  
**Type:** character vector  
**Values:** 'auto' | 'real' | 'complex'  
**Default:** 'auto'

**Sample time — Specify sample time as a value other than -1**
-1 (default) | scalar | vector

Specify the sample time as a value other than -1. For more information, see “Specify Sample Time”.
Dependencies

This parameter is not visible unless it is explicitly set to a value other than -1. To learn more, see “Blocks for Which Sample Time Is Not Recommended”.

Programmatic Use
Block Parameter: SampleTime
Type: character vector
Values: scalar or vector
Default: '-1'

Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>double</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

a. This block supports fixed-point and base integer data types for 'Approximation method' CORDIC.

More About

CORDIC

CORDIC is an acronym for COordinate Rotation DIGital Computer. The Givens rotation-based CORDIC algorithm is one of the most hardware-efficient algorithms available because it requires only iterative shift-add operations (see References). The CORDIC algorithm eliminates the need for explicit multipliers. Using CORDIC, you can calculate various functions, such as sine, cosine, arc sine, arc cosine, arc tangent, and vector magnitude. You can also use this algorithm for divide, square root, hyperbolic, and logarithmic functions.
Increasing the number of CORDIC iterations can produce more accurate results, but doing so also increases the expense of the computation and adds latency.

References


Extended Capabilities

**C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

Not all compilers support the asinh, acosh, and atanh functions. If you use a compiler that does not support those functions, a warning appears and the generated code fails to link.

**HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.
**HDL Architecture**

You can generate HDL code for all functions of the block with single data types and architecture set to Trigonometric in native floating-point mode. The native floating-point mode does not support double data types for the block.

This block has multi-cycle implementations that introduce additional latency in the generated code. To see the added latency, view the generated model or validation model. See “Generated Model and Validation Model” (HDL Coder).

The Trigonometric Function block supports HDL code generation for these functions in this table with CORDIC approximation method and fixed-point data types.

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Function</th>
<th>Approximation Method</th>
<th>Use Pipelined Kernel Setting</th>
<th>Additional cycles of latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>atan2</td>
<td>atan2</td>
<td>CORDIC</td>
<td>On</td>
<td>Number of iterations + 3. For example, if you specify <strong>Number of iterations</strong> as 19, the code generator inserts a delay of length 22 at the output.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Off</td>
<td>0</td>
</tr>
<tr>
<td>SinCosCordic</td>
<td>sin</td>
<td>CORDIC</td>
<td>On</td>
<td>Number of iterations + 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Off</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>cos</td>
<td>CORDIC</td>
<td>On</td>
<td>Number of iterations + 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Off</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>cos + jsin</td>
<td>CORDIC</td>
<td>On</td>
<td>Number of iterations + 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Off</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>sin cos</td>
<td>CORDIC</td>
<td>On</td>
<td>Number of iterations + 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Off</td>
<td>0</td>
</tr>
</tbody>
</table>
### HDL Block Properties

#### General

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConstrainedOutputPipeline</td>
<td>Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. For more details, see “ConstrainedOutputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td>InputPipeline</td>
<td>Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “InputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td>OutputPipeline</td>
<td>Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “OutputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td>UsePipelinedKernel</td>
<td>Whether to use a pipelined implementation of the CORDIC algorithm in the generated code. The default is On.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Setting</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>On (default)</td>
<td>Use a pipelined implementation of the CORDIC algorithm. The pipelined implementation adds latency.</td>
</tr>
<tr>
<td>Off</td>
<td>Use a combinatorial implementation of the CORDIC algorithm. The combinatorial implementation does not add latency. If the block is in a feedback loop, use this implementation.</td>
</tr>
</tbody>
</table>

#### Native Floating Point

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>InputRangeReduction</td>
<td>Use this property for the sin, cos, tan, sincos, and cos+jsin functions. If your input range is unbounded, enable this property for HDL Coder to insert additional logic to reduce the range of inputs to [-pi, pi]. See also “InputRangeReduction” (HDL Coder).</td>
</tr>
</tbody>
</table>
**Native Floating Point**

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HandleDenormals</td>
<td>Specify whether you want HDL Coder to insert additional logic to handle denormal numbers in your design. Denormal numbers are numbers that have magnitudes less than the smallest floating-point number that can be represented without leading zeros in the mantissa. The default is inherit. See also “HandleDenormals” (HDL Coder).</td>
</tr>
<tr>
<td>LatencyStrategy</td>
<td>Specify whether to map the blocks in your design to inherit, Max, Min, or Zero for the floating-point operator. The default is inherit. See also “LatencyStrategy” (HDL Coder).</td>
</tr>
<tr>
<td>MultiplyStrategy</td>
<td>Use this property for the sin, cos, tan, sincos, and cos+jsin functions. The default is inherit. Specify whether you want to use FullMultiplier or PartMultiplierPartAddShift. See also “MantissaMultiplyStrategy” (HDL Coder).</td>
</tr>
</tbody>
</table>

**Restrictions**

- For the sin and cos functions, only signed fixed-point data types are supported for CORDIC approximations.
- HDL Coder displays an error when you select these settings for a Trigonometric Function block inside a feedback loop:
  - HDL architecture as SinCosCordic
  - UsePipelinedKernel as On

The error occurs because the block is in a feedback loop and the code generator is unable to insert additional latency. To avoid this error, add a delay of length equal to the Number of iterations + 3 adjacent to the block. The code generator then absorbs this delay to meet the additional latency of the Trigonometric Function block.

For example, this Trigonometric Function block has Number of iterations set to 30. A Delay of length 33 adjacent to the block meets the additional latency.
PLC Code Generation
Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

This block supports fixed-point and base integer data types when you set the Function to sin, cos, sincos, cos + jsin, or atan2, and set the Approximation method to CORDIC.

See Also

Blocks
Math Function | Sine, Cosine | Sqrt

Functions
cordiccos | cordicsin | cordicsincos

Introduced before R2006a

1-2216
Unary Minus

Negate input

Library:
- Simulink / Math Operations
- HDL Coder / HDL Floating Point Operations
- HDL Coder / Math Operations

Description

The Unary Minus block negates the input.

Ports

Input

Port_1 — Signal to negate
- scalar | vector | matrix | N-D array

Input signal, specified as a scalar, vector, matrix, or N-D array.

Data Types: single | double | int8 | int16 | int32 | fixed point

Output

Port_1 — Negation of input signal
- scalar | vector | matrix | N-D array

Negation of the input signal. The output has the same data type and dimensions as the input.

Data Types: single | double | int8 | int16 | int32 | fixed point
Parameters

Saturate on integer overflow — Method of overflow action

off (default) | on

Select to have integer overflows saturate. Otherwise, overflows wrap.

When you select this check box, saturation applies to every internal operation on the block, not just the output, or result. In general, the code generation process can detect when overflow is not possible. In this case, the code generator does not produce saturation code.

For signed-integer data types, the unary minus of the most negative value is not representable by the data type. In this case, the Saturate on integer overflow check box controls the behavior of the block:

<table>
<thead>
<tr>
<th>Parameter Setting</th>
<th>Block Behavior</th>
<th>Examples</th>
</tr>
</thead>
</table>
| Saturate on integer overflow = on | Values saturate to the most positive value of the integer data type | • For 8-bit signed integers, -128 maps to 127.  
• For 16-bit signed integers, -32768 maps to 32767.  
• For 32-bit signed integers, -2147483648 maps to 2147483647. |
| Saturate on integer overflow = off | Values wrap to the most negative value of the integer data type | • For 8-bit signed integers, -128 remains -128.  
• For 16-bit signed integers, -32768 remains -32768.  
• For 32-bit signed integers, -2147483648 remains -2147483648. |

Programmatic Use

Block Parameter: SaturateOnIntegerOverflow
Type: character vector
Values: 'off' | 'on'
Default: 'off'

Sample time — Specify sample time as a value other than -1
-1 (default) | scalar | vector
Specify the sample time as a value other than -1. For more information, see “Specify Sample Time”.

**Dependencies**

This parameter is not visible unless it is explicitly set to a value other than -1. To learn more, see “Blocks for Which Sample Time Is Not Recommended”.

**Programmatic Use**

**Block Parameter:** SampleTime  
**Type:** character vector  
**Values:** scalar or vector  
**Default:** '-1'

**Block Characteristics**

<table>
<thead>
<tr>
<th>Data Types</th>
<th>double</th>
<th>fixed point(^a)</th>
<th>integer(^a)</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\(^a\) This block only supports signed fixed-point data types.

**Extended Capabilities**

**C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.
HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture
This block has a single, default HDL architecture.

HDL Block Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConstrainedOutput</td>
<td>Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. For more details, see “ConstrainedOutputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td>Pipeline</td>
<td></td>
</tr>
<tr>
<td>InputPipeline</td>
<td>Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “InputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td>OutputPipeline</td>
<td>Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “OutputPipeline” (HDL Coder).</td>
</tr>
</tbody>
</table>

Complex Data Support
This block supports code generation for complex signals.

PLC Code Generation
Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

This block only supports signed fixed-point data types.
See Also
uminus

Topics
“Operator Precedence” (MATLAB)

Introduced before R2006a
Uniform Random Number

Generate uniformly distributed random numbers

Library: Simulink / Sources

Description

The Uniform Random Number block generates uniformly distributed random numbers over an interval that you specify. To generate normally distributed random numbers, use the Random Number block.

You can generate a repeatable sequence using any Uniform Random Number block with the same nonnegative seed and parameters. The seed resets to the specified value each time a simulation starts.

Avoid integrating a random signal, because solvers must integrate relatively smooth signals. Instead, use the Band-Limited White Noise block.

The numeric parameters of this block must have the same dimensions after scalar expansion. If you select the Interpret vector parameters as 1-D check box and the numeric parameters are row or column vectors after scalar expansion, the block outputs a 1-D signal. If you clear the Interpret vector parameters as 1-D check box, the block outputs a signal of the same dimensionality as the parameters.

Ports

Output

Port_1 — Random number output signal
scalar | vector

Output signal of generated uniformly distributed random numbers over the interval you specify.
Data Types: double

## Parameters

### Minimum — Minimum interval

Specify the minimum of the interval.

**Programmatic Use**

**Block Parameter:** Minimum  
**Type:** character vector  
**Values:** scalar  
**Default:** ' -1'

### Maximum — Maximum interval

Specify the maximum of the interval.

**Programmatic Use**

**Block Parameter:** Maximum  
**Type:** character vector  
**Values:** scalar  
**Default:** ' 1'

### Seed — Random number seed

Specify the starting seed for the random number generator.

The seed must be 0 or a positive integer. Output is repeatable for a given seed.

**Programmatic Use**

**Block Parameter:** See  
**Type:** character vector  
**Values:** scalar  
**Default:** ' 0'

### Sample time — Sample time

Sample time

0.1 (default) | scalar
Specify the time interval between samples. See “Specify Sample Time” in the Simulink documentation for more information.

**Programmatic Use**

**Block Parameter:** SampleTime

**Type:** character vector

**Values:** scalar

**Default:** '0.1'

**Interpret vector parameters as 1-D — Treat vectors as 1-D**

- Select this check box to output a vector of length \( N \) if the **Constant value** parameter evaluates to an \( N \)-element row or column vector.

  - When you select this check box, the block outputs a vector of length \( N \) if the **Constant value** parameter evaluates to an \( N \)-element row or column vector. For example, the block outputs a matrix of dimension 1-by-\( N \) or \( N \)-by-1.

  - When you clear this check box, the block does not output a vector of length \( N \) if the **Constant value** parameter evaluates to an \( N \)-element row or column vector.

**Programmatic Use**

**Block Parameter:** VectorParams1D

**Type:** character vector

**Values:** 'on' | 'off'

**Default:** 'on'

### Block Characteristics

<table>
<thead>
<tr>
<th><strong>Data Types</strong></th>
<th>double</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Direct Feedthrough</strong></td>
<td>no</td>
</tr>
<tr>
<td><strong>Multidimensional Signals</strong></td>
<td>yes</td>
</tr>
<tr>
<td><strong>Variable-Size Signals</strong></td>
<td>no</td>
</tr>
</tbody>
</table>
Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also
Random Number | Repeating Sequence

Introduced before R2006a
Unit System Configuration

Configure units
Library: Simulink / Ports & Subsystems

Description

The Unit System Configuration block specifies allowed and disallowed unit systems for the component. It restricts units systems for a subsystem or top model and all its children, unless you override it with another Unit System Configuration block in a child.

This block supports normal, accelerator, and rapid accelerator modes and fast restart.

Parameters

Disallowed unit systems — Disallowed unit systems
SI | English | SI (extended) | CGS

Displays a list of the disallowed unit systems. By default, the Allow all unit systems check box is selected, and all unit systems are allowed.

To designate a unit system as disallowed, select it in the Allowed unit systems column and click << Disallow.

Dependencies

To enable changes to this parameter, you must first clear the Allow all unit systems check box.

Allowed unit systems — Allowed unit systems
SI | English | SI (extended) | CGS

Displays a list of the allowed unit systems. By default, the Allow all unit systems check box is selected, and all unit systems are allowed.
To designate a unit system as allowed, use the Allow >> and << Disallow buttons to move unit systems between the Disallowed unit systems and Allowed unit systems columns.

**Dependencies**

To enable changes to this parameter, you must first clear the Allow all unit systems check box.

**Programmatic Use**

**Block Parameter:** UnitSystems  
**Type:** cell array of character vectors  
**Values:** cell array of the following character vectors: 'SI' | 'English' | 'SI (extended)' | 'CGS'  
**Default:** {'SI', 'English', 'SI (extended)', 'CGS'}

**Allow all unit systems — Allow all unit systems**  
_on (default) | off_

When you select this check box, all unit systems are allowed. To restrict the allowed unit systems to only the ones specified in the Allowed unit systems column, clear this check box.

**Dependencies**

Selecting the Allow all unit systems check box disables the Disallowed unit systems and Allowed unit systems parameters.

**Programmatic Use**

**Block Parameter:** AllowAllUnitSystems  
**Type:** character vector  
**Values:** 'on' | 'off'  
**Default:** 'on'

**Block Characteristics**

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>bus</th>
<th>double</th>
<th>enumerated</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1-2227
### Extended Capabilities

#### C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

#### Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

#### See Also
Inport | Outport | Unit Conversion

#### Topics
“Update an Existing Model to Use Units”
“Units in Simulink”
“Unit Specification in Simulink Models”
“Restricting Unit Systems”

**Introduced in R2016a**
Unit Conversion

Convert units

Library: Simulink / Signal Attributes

Description

The Unit Conversion block converts the unit of the input signal to the output signal. The block can perform the conversion if the units are separated by a scaling factor or offset, or are inverse units, for example:

- \( y = a \cdot U \)
- \( y = a \cdot U + b \), where \( a \) is the scale and \( b \) is the offset
- \( y = a/U \)

This block supports normal, accelerator, and rapid accelerator modes and fast restart.

Ports

Input

Port_1 — Input signal

scalar | vector | matrix

Input signal for unit conversion, specified as a scalar, vector, or matrix.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

Output

Port_1 — Output signal

scalar | vector | matrix
Output signal with converted units, specified as a scalar, vector, or matrix.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | fixed point

**Parameters**

**Output data type — Output data type**

*Inherit: Inherit via internal rule (default)* | *Inherit: Inherit via back propagation*

Specify the output data type.

- **Inherit: Inherit via internal rule** — Simulink chooses intermediate and output data types to balance numerical accuracy, performance, and generated code size, while accounting for the properties of the embedded target hardware. If you change the embedded target settings, the data type selected by the internal rule might change.

- **Inherit: Inherit via back propagation** — Output data type is inherited via back propagation. Internal rules determine the intermediate data types and Simulink casts the final results to the output data type.

**Programmatic Use**

**Block Parameter:** `OutDataTypeStr`

**Type:** character vector

**Values:** `'Inherit via internal rule'` | `'Inherit via back propagation'`

**Default:** `'Inherit via internal rule'`

**Block Characteristics**

<table>
<thead>
<tr>
<th>Data Types</th>
<th>double</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

See Also
“Converting Units”

Topics
“Units in Simulink”
“Converting Units”

Introduced in R2016a
Unit Delay

Delay signal one sample period
Library: Simulink / Discrete
HDL Coder / Discrete

Description

The Unit Delay block holds and delays its input by the sample period you specify. When placed in an iterator subsystem, it holds and delays its input by one iteration. This block is equivalent to the $z^{-1}$ discrete-time operator. The block accepts one input and generates one output. Each signal can be scalar or vector. If the input is a vector, the block holds and delays all elements of the vector by the same sample period.

You specify the block output for the first sampling period with the Initial conditions parameter. Careful selection of this parameter can minimize unwanted output behavior. You specify the time between samples with the Sample time parameter. A setting of -1 means the block inherits the Sample time.

Note The Unit Delay block errors out if you use it to create a transition between blocks operating at different sample rates. Use the Rate Transition block instead.

Comparison with Similar Blocks

The Memory, Unit Delay, and Zero-Order Hold blocks provide similar functionality but have different capabilities. Also, the purpose of each block is different.

This table shows recommended usage for each block.
Block | Purpose of the Block | Reference Examples
--- | --- | ---
Unit Delay | Implement a delay using a discrete sample time that you specify. The block accepts and outputs signals with a discrete sample time. | • sldemo_enginewc (Compression subsystem)

Memory on page 1-1278 | Implement a delay by one major integration time step. Ideally, the block accepts continuous (or fixed in minor time step) signals and outputs a signal that is fixed in minor time step. | • sldemo_bounce
• sldemo_clutch (Friction Mode Logic/Lockup FSM subsystem)

Zero-Order Hold | Convert an input signal with a continuous sample time to an output signal with a discrete sample time. | • sldemo_radar_eml
• aero_dap3dof

Each block has the following capabilities.

<table>
<thead>
<tr>
<th>Capability</th>
<th>Memory</th>
<th>Unit Delay</th>
<th>Zero-Order Hold</th>
</tr>
</thead>
<tbody>
<tr>
<td>Specification of initial condition</td>
<td>Yes</td>
<td>Yes</td>
<td>No, because the block output at time t = 0 must match the input value.</td>
</tr>
<tr>
<td>Specification of sample time</td>
<td>No, because the block can only inherit sample time from the driving block or the solver used for the entire model.</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Support for frame-based signals</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Support for state logging</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>
Ports

Input

Port_1 — Input signal
scalar | vector

Input signal that the block delays by one sample period.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus

Output

Port_1 — Output signal
scalar | vector

Output signal that is the input delayed by one sample period.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus

Parameters

Main

Initial condition — First sample period output
θ (default) | scalar | vector

Specify the output of the simulation for the first sampling period, during which the output of the Unit Delay block is otherwise undefined.

Programmatic Use
Block Parameter: InitialCondition
Type: character vector
Value: scalar | vector
Default: 'θ'

1-2234
**Input processing — Specify sample- or frame-based processing**

Elements as channels (sample based) (default) | Columns as channels (frame based)

Specify whether the block performs sample- or frame-based processing:

- **Columns as channels (frame based)** — Treat each column of the input as a separate channel (frame-based processing).

**Note** Frame-based processing requires a DSP System Toolbox license.

For more information, see “Sample- and Frame-Based Concepts” (DSP System Toolbox).

- **Elements as channels (sample based)** — Treat each element of the input as a separate channel (sample-based processing).

Use **Input processing** to specify whether the block performs sample- or frame-based processing. The block accepts frame-based signals for the input u. All other input signals must be sample-based.

<table>
<thead>
<tr>
<th>Input Signal u</th>
<th>Input Processing Mode</th>
<th>Block Works?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sample based</td>
<td>Sample based</td>
<td>Yes</td>
</tr>
<tr>
<td>Frame based</td>
<td></td>
<td>No, produces an error</td>
</tr>
<tr>
<td>Sample based</td>
<td>Frame based</td>
<td>Yes</td>
</tr>
<tr>
<td>Frame based</td>
<td></td>
<td>Yes</td>
</tr>
</tbody>
</table>

For more information about these two processing modes, see “Sample- and Frame-Based Concepts” (DSP System Toolbox).

**Programmatic Use**

**Block Parameter:** InputProcessing  
**Type:** character vector  
**Values:** 'Columns as channels (frame based)' | 'Elements as channels (sample based)'  
**Default:** 'Elements as channels (sample based)'

**Sample time (-1 for inherited) — Discrete interval between sample time hits**

-1 (default) | scalar
Enter the discrete interval between sample time hits or specify -1 to inherit the sample time.

See also “Specify Sample Time”.

**Programmatic Use**
**Block Parameter:** SampleTime  
**Type:** character vector  
**Value:** real scalar  
**Default:** '-1'

**State Attributes**

**State name — Unique name for block state**  
' ' (default) | alphanumeric string

Use this parameter to assign a unique name to the block state. The default is ' '. When this field is blank, no name is assigned. When using this parameter, remember these considerations:

- A valid identifier starts with an alphabetic or underscore character, followed by alphanumeric or underscore characters.
- The state name applies only to the selected block.

This parameter enables **State name must resolve to Simulink signal object** when you click **Apply**.

For more information, see “Apply Storage Classes to Individual Signal, State, and Parameter Data Elements” (Simulink Coder).

**Programmatic Use**
**Block Parameter:** StateName  
**Type:** character vector  
**Values:** unique name  
**Default:** ' '

**State name must resolve to Simulink signal object — Require state names resolve to signal object**  
**Off** (default) | **Boolean**
Specify if requiring that state name resolve to Simulink signal objects or not. If selected, the software generates an error at run time if you specify a state name that does not match the name of a Simulink signal object.

**Dependency**

Enabled when you give the parameter **State name** a value and set the model configuration parameter **Signal resolution** to a value other than **None**.

Selecting this check box disables **Code generation storage class**.

**Programmatic Use**

**Block Parameter:** StateMustResolveToSignalObject  
**Type:** character vector  
**Values:** 'off' | 'on'  
**Default:** 'off'

**Signal object class — Custom storage class package name**

Choose a custom storage class package by selecting a signal object class that the target package defines. For example, to apply custom storage classes from the built-in package `mpt`, select `mpt.Signal`. Unless you use an ERT-based code generation target with Embedded Coder, custom storage classes do not affect the generated code.

If the class that you want does not appear in the list, select **Customize class lists**. For instructions, see “Target Class Does Not Appear in List of Signal Object Classes” (Embedded Coder).

For information about storage classes, see “Apply Storage Classes to Individual Signal, State, and Parameter Data Elements” (Simulink Coder). For information about custom storage classes, see “Apply Built-In and Customized Storage Classes to Data Elements” (Embedded Coder).

**Programmatic Use**

**Block Parameter:** StateSignalObject  
**Type:** character vector  
**Values:** 'Simulink.Signal' | '<StorageClass.PackageName>'  
**Default:** 'Simulink.Signal'

**Code generation storage class — Storage class for code generation**

Auto (default) | Model default | ExportedGlobal | ImportedExtern | ImportedExternPointer | Bitfield (Custom) | Volatile (Custom) |
Select state storage class for code generation. If you do not need to interface to external code, select Auto.

For more information, see “Apply Storage Classes to Individual Signal, State, and Parameter Data Elements” (Simulink Coder) and “Apply Built-In and Customized Storage Classes to Data Elements” (Embedded Coder).

**Programmatic Use**

**Block Parameter:** StateStorageClass

**Type:** character vector

**Values:** 'Auto' | 'Model default' | 'ExportedGlobal' | 'ImportedExtern' | 'ImportedExternPointer' | 'Custom'

**Default:** 'Auto'

**Block Characteristics**

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>bus</th>
<th>double</th>
<th>enumerated</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Direct Feedthrough</th>
<th>yes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multidimensional Signals</td>
<td>yes</td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>yes</td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
</tr>
</tbody>
</table>

**Extended Capabilities**

**C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.
Generated code relies on memcpy or memset functions (strong.h) under certain conditions.

**HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

**HDL Architecture**

This block has a single, default HDL architecture.

**HDL Block Properties**

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>InputPipeline</td>
<td>Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “InputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td>OutputPipeline</td>
<td>Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “OutputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td>ResetType</td>
<td>Suppress reset logic generation. The default is default, which generates reset logic. See also “ResetType” (HDL Coder).</td>
</tr>
</tbody>
</table>

**Complex Data Support**

This block supports code generation for complex signals.

**PLC Code Generation**

Generate Structured Text code using Simulink® PLC Coder™.

**Fixed-Point Conversion**

Design and simulate fixed-point systems using Fixed-Point Designer™.
See Also
Delay | Resettable Delay

Introduced before R2006a
Variable Integer Delay

Delay input signal by variable sample period

Library: Simulink / Discrete

Description

The Variable Integer Delay block is a variant of the Delay block that has the source of the delay length set to Input port, by default.
Ports

Input

\( u \) — Data input signal
scalar | vector

Input data signal delayed according to parameters settings.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus

\( d \) — Delay length
scalar

Delay length specified as inherited from an input port. Enabled when you select the Delay length: Source parameter as Input port.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

Enable — External enable signal
scalar

Enable signal that enables or disables execution of the block. To create this port, select the Show enable port parameter.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point

External reset — External reset signal
scalar

External signal that resets execution of the block to the initial condition. To create this port, select the External reset parameter.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point

\( x_0 \) — Initial condition
scalar | vector
Initial condition specified as inherited from an input port. Enabled when you select the Initial Condition: Source parameter as Input port.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

Output

Port_1 — Output signal
scalar | vector

Output signal that is the input signal delayed by the length of time specified by the parameter Delay length. The initial value of the output signal depends on several conditions. See “Initial Block Output” on page 1-360.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus

Parameters

Main

Delay length — Delay length
Dialog (default) | Input port

Specify whether to enter the delay length directly on the dialog box (fixed delay) or to inherit the delay from an input port (variable delay).

- If you set Source to Dialog, enter the delay length in the edit field under Value.
- If you set Source to Input port, verify that an upstream signal supplies a delay length for the input port. You can also specify its maximum value by specifying the parameter Upper limit.

Specify the scalar delay length as a real, non-negative integer. An out-of-range or non-integer value in the dialog box (fixed delay) returns an error. An out-of-range value from an input port (variable delay) casts it into the range. A non-integer value from an input port (variable delay) truncates it to the integer.
Programmatic Use

Block Parameter: DelayLengthSource
Type: character vector
Values: 'Dialog' | 'Input port'
Default: 'Dialog'

Block Parameter: DelayLength
Type: character vector
Values: scalar
Default: '2'

Block Parameter: DelayLengthUpperLimit
Type: character vector
Values: scalar
Default: '100'

Initial condition — Initial condition

Specify whether to enter the initial condition directly on the dialog box or to inherit the initial condition from an input port.

- If you set Source to Dialog, enter the initial condition in the edit field under Value.
- If you set Source to Input port, verify that an upstream signal supplies an initial condition for the x0 input port.

Simulink converts offline the data type of Initial condition to the data type of the input signal u using a round-to-nearest operation and saturation.

Note When State name must resolve to Simulink signal object is selected on the State Attributes pane, the block copies the initial value of the signal object to the Initial condition parameter. However, when the source for Initial condition is Input port, the block ignores the initial value of the signal object.

Programmatic Use

Block Parameter: InitialConditionSource
Type: character vector
Values: 'Dialog' | 'Input port'
Default: 'Dialog'

Block Parameter: InitialCondition
Type: character vector
**Values:** scalar  
**Default:** '0.0'

**Input processing — Specify sample- or frame-based processing**  
Elements as channels (sample based) (default) | Columns as channels (frame based)

Specify whether the block performs sample- or frame-based processing:

- **Columns as channels (frame based)** — Treat each column of the input as a separate channel (frame-based processing).

  **Note**  
  Frame-based processing requires a DSP System Toolbox license.

  For more information, see “Sample- and Frame-Based Concepts” (DSP System Toolbox).

- **Elements as channels (sample based)** — Treat each element of the input as a separate channel (sample-based processing).

Use **Input processing** to specify whether the block performs sample- or frame-based processing. The block accepts frame-based signals for the input \( u \). All other input signals must be sample-based.

<table>
<thead>
<tr>
<th>Input Signal u</th>
<th>Input Processing Mode</th>
<th>Block Works?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sample based</td>
<td>Sample based</td>
<td>Yes</td>
</tr>
<tr>
<td>Frame based</td>
<td></td>
<td>No, produces an error</td>
</tr>
<tr>
<td>Sample based</td>
<td>Frame based</td>
<td>Yes</td>
</tr>
<tr>
<td>Frame based</td>
<td></td>
<td>Yes</td>
</tr>
</tbody>
</table>

For more information about these two processing modes, see “Sample- and Frame-Based Concepts” (DSP System Toolbox).

**Programmatic Use**  
**Block Parameter:** InputProcessing  
**Type:** character vector  
**Values:** 'Columns as channels (frame based)' | 'Elements as channels (sample based)'  
**Default:** 'Elements as channels (sample based)'
**Use circular buffer for state — Circular buffer for storing state**

off (default) | on

Select to use a circular buffer for storing the state in simulation and code generation. Otherwise, an array buffer stores the state.

Using a circular buffer can improve execution speed when the delay length is large. For an array buffer, the number of copy operations increases as the delay length goes up. For a circular buffer, the number of copy operations is constant for increasing delay length.

If one of the following conditions is true, an array buffer always stores the state because a circular buffer does not improve execution speed:

- For sample-based signals, the delay length is 1.
- For frame-based signals, the delay length is no larger than the frame size.

**Programmatic Use**

Block Parameter: `UseCircularBuffer`

Type: character vector

Values: 'off' | 'on'

Default: 'off'

**Prevent direct feedthrough — Prevent direct feedthrough**

off (default) | on

Select to increase the delay length from zero to the lower limit for the **Input processing** mode:

- For sample-based signals, increase the minimum delay length to 1.
- For frame-based signals, increase the minimum delay length to the frame length.

Selecting this check box prevents direct feedthrough from the input port, \( u \), to the output port. However, this check box cannot prevent direct feedthrough from the initial condition port, \( x_0 \), to the output port.

**Dependency**

To enable this parameter, set **Delay length: Source** to **Input port**.

**Programmatic Use**

Block Parameter: `PreventDirectFeedthrough`

Type: character vector
Values: 'off' | 'on'
Default: 'off'

Remove delay length check in generated code — Remove delay length out-of-range check
off (default) | on

Select to remove code that checks for out-of-range delay length.

<table>
<thead>
<tr>
<th>Check Box</th>
<th>Result</th>
<th>When to Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selected</td>
<td>Generated code does not include conditional statements to check for out-of-range delay length.</td>
<td>For code efficiency</td>
</tr>
<tr>
<td>Cleared</td>
<td>Generated code includes conditional statements to check for out-of-range delay length.</td>
<td>For safety-critical applications</td>
</tr>
</tbody>
</table>

Dependency

To enable this parameter, set **Delay length: Source** to **Input port**.

Programmatic Use

**Block Parameter**: RemoveDelayLengthCheckInGeneratedCode

**Type**: character vector

**Values**: 'off' | 'on'

**Default**: 'off'

Diagnostic for delay length — Diagnostic checks for delay length

None (default) | Warning | Error

Specify whether to produce a warning or error when the input d is less than the lower limit or greater than the **Delay length: Upper limit**. The lower limit depends on the setting for **Prevent direct feedthrough**.

- If the check box is cleared, the lower limit is zero.
- If the check box is selected, the lower limit is 1 for sample-based signals and frame length for frame-based signals.

Options for the diagnostic include:
- **None** — Simulink software takes no action.
- **Warning** — Simulink software displays a warning and continues the simulation.
- **Error** — Simulink software terminates the simulation and displays an error.

**Dependency**

To enable this parameter, set **Delay length:** **Source** to **Input** **port**.

**Programmatic Use**

**Block Parameter:** DiagnosticForDelayLength

**Type:** character vector

**Values:** 'None' | 'Warning' | 'Error'

**Default:** 'None'

**Show enable port — Create enable port**

**off** (default) | **on**

Select to control execution of this block with an enable port. The block is considered enabled when the input to this port is nonzero, and is disabled when the input is 0. The value of the input is checked at the same time step as the block execution.

**Programmatic Use**

**Block Parameter:** ShowEnablePort

**Type:** character vector

**Values:** 'off' | 'on'

**Default:** 'off'

**External reset — External state reset**

**None** (default) | **Rising** | **Falling** | **Either** | **Level** | **Level hold**

Specify the trigger event to use to reset the states to the initial conditions.

<table>
<thead>
<tr>
<th>Reset Mode</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>No reset</td>
</tr>
<tr>
<td>Rising</td>
<td>Reset on a rising edge</td>
</tr>
<tr>
<td>Falling</td>
<td>Reset on a falling edge</td>
</tr>
<tr>
<td>Either</td>
<td>Reset on either a rising or falling edge</td>
</tr>
</tbody>
</table>
Reset Mode | Behavior
---|---
Level | Reset in either of these cases:
  • When the reset signal is nonzero at the current time step
  • When the reset signal value changes from nonzero at the previous time step to zero at the current time step
Level hold | Reset when the reset signal is nonzero at the current time step

**Programmatic Use**

**Block Parameter:** ExternalReset
**Type:** character vector
**Values:** 'None' | 'Rising' | 'Falling' | 'Either' | 'Level' | 'Level hold'
**Default:** 'None'

**Sample time (-1 for inherited) — Discrete interval between sample time hits**
-1 (default) | scalar

Specify the time interval between samples. To inherit the sample time, set this parameter to -1. This block supports discrete sample time, but not continuous sample time.

**Programmatic Use**

**Block Parameter:** SampleTime
**Type:** character vector
**Value:** real scalar
**Default:** ' -1'

**State Attributes**

**State name — Unique name for block state**
' ' (default) | alphanumeric string

Use this parameter to assign a unique name to the block state. The default is ' '. When this field is blank, no name is assigned. When using this parameter, remember these considerations:

• A valid identifier starts with an alphabetic or underscore character, followed by alphanumeric or underscore characters.
The state name applies only to the selected block. This parameter enables **State name must resolve to Simulink signal object** when you click **Apply**.

For more information, see “Apply Storage Classes to Individual Signal, State, and Parameter Data Elements” (Simulink Coder).

**Programmatic Use**
**Block Parameter:** `StateName`
**Type:** character vector
**Values:** unique name
**Default:** ''

**State name must resolve to Simulink signal object — Require state name resolve to a signal object**
**off** (default) | **on**

Select this check box to require that the state name resolves to a Simulink signal object.

**Dependencies**
To enable this parameter, specify a value for **State name**. This parameter appears only if you set the model configuration parameter **Signal resolution** to a value other than **None**.

Selecting this check box disables **Code generation storage class**.

**Programmatic Use**
**Block Parameter:** `StateMustResolveToSignalObject`
**Type:** character vector
**Values:** 'off' | 'on'
**Default:** 'off'

**Signal object class — Custom storage class package name**
**Simulink.Signal** (default) | `<StorageClass.PackageName>`

Choose a custom storage class package by selecting a signal object class that the target package defines. For example, to apply custom storage classes from the built-in package `mpt`, select `mpt.Signal`. Unless you use an ERT-based code generation target with Embedded Coder, custom storage classes do not affect the generated code.

If the class that you want does not appear in the list, select **Customize class lists**. For instructions, see “Target Class Does Not Appear in List of Signal Object Classes” (Embedded Coder).
For information about storage classes, see “Apply Storage Classes to Individual Signal, State, and Parameter Data Elements” (Simulink Coder). For information about custom storage classes, see “Apply Built-In and Customized Storage Classes to Data Elements” (Embedded Coder).

**Programmatic Use**

**Block Parameter:** StateSignalObject  
**Type:** character vector  
**Values:** 'Simulink.Signal' | '<StorageClass.PackageName>'  
**Default:** 'Simulink.Signal'

**Code generation storage class — State storage class for code generation**

Auto (default) | Model default | ExportedGlobal | ImportedExtern | ImportedExternPointer | BitField (Custom) | Model default | ExportToFile (Custom) | ImportFromFile (Custom) | FileScope (Custom) | AutoScope (Custom) | Struct (Custom) | GetSet (Custom) | Reusable (Custom)

Select state storage class for code generation.

- **Auto** is the appropriate storage class for states that you do not need to interface to external code.
- **StorageClass** applies the storage class or custom storage class that you select from the list. For information about storage classes, see “Apply Storage Classes to Individual Signal, State, and Parameter Data Elements” (Simulink Coder). For information about custom storage classes, see “Apply Built-In and Customized Storage Classes to Data Elements” (Embedded Coder).

Use **Signal object class** to select custom storage classes from a package other than Simulink.

**Dependencies**

To enable this parameter, specify a value for **State name**.

**Programmatic Use**

**Block Parameter:** StateStorageClass  
**Type:** character vector  
**Values:** 'Auto' | 'Model default' | 'ExportedGlobal' | 'ImportedExtern' | 'ImportedExternPointer' | 'Custom' | ...  
**Default:** 'Auto'
Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>bus</th>
<th>double</th>
<th>enumerated</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
</table>

| Direct Feedthrough            | yes     |
| Multidimensional Signals      | yes     |
| Variable-Size Signals         | yes     |
| Zero-Crossing Detection       | no      |

Extended Capabilities

**C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

Generated code relies on memcpy or memset functions (string.h) under certain conditions.

**HDL Code Generation**

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

For information about HDL code generation, see “HDL Code Generation” on page 1-373 on the Delay page.

**Fixed-Point Conversion**

Design and simulate fixed-point systems using Fixed-Point Designer™.
See Also
Delay | Resettable Delay | Tapped Delay | Unit Delay

Topics
“Using Enabled Subsystems”

Introduced in R2012b
Variable Time Delay

Delay input by variable amount of time

Library: Simulink / Continuous

Description

The Variable Transport Delay and Variable Time Delay blocks appear as two blocks in the Simulink block library. However, they are the same Simulink block with different settings for the Select delay type parameter. Use this parameter to specify the mode in which the block operates.

Variable Time Delay

In this mode, the block has a data input, a time delay input, and a data output. (See “Port Location After Rotating or Flipping” for a description of the port order for various block orientations.) The output at the current time step equals the value of its data input at a previous time step. This time step is the current simulation time minus a delay time specified by the time delay input.

\[ y(t) = u(t - t_0) = u(t - \tau(t)) \]

During the simulation, the block stores time and input value pairs in an internal buffer. At the start of simulation, the block outputs the value of the Initial output parameter until the simulation time exceeds the time delay input. Then, at each simulation step, the block outputs the signal at the time that corresponds to the current simulation time minus the delay time.

If you want the output at a time between input storing times and the solver is a continuous solver, the block interpolates linearly between points. If the time delay is
smaller than the step size, the block extrapolates an output point from a previous point. For example, consider a fixed-step simulation with a step size of 1 and the current time at \( t = 5 \). If the delay is 0.5, the block must generate a point at \( t = 4.5 \), but the most recent stored time value is at \( t = 4 \). Thus, the block extrapolates the input at 4.5 from the input at 4 and uses the extrapolated value as its output at \( t = 5 \).

Extrapolating forward from the previous time step can produce a less accurate result than extrapolating back from the current time step. However, the block cannot use the current input to calculate its output value because the input port does not have direct feedthrough.

If the model specifies a discrete solver, the block does not interpolate between time steps. Instead, it returns the nearest stored value that precedes the required value.

**Variable Transport Delay**

In this mode, the block output at the current time step is equal to the value of its data (top, or left) input at an earlier time step equal to the current time minus a transport delay.

\[
y(t) = u(t - t_d(t))
\]

Simulink software finds the transport delay, \( t_d(t) \), by solving the following equation:

\[
\int_{t - t_d(t)}^{t} \frac{1}{t_i(\tau)} d\tau = 1
\]

This equation involves an instantaneous time delay, \( t_i(t) \), given by the time delay (bottom, or right) input.

Suppose that you want to use this block to model the fluid flow through a pipe where the fluid speed varies with time. In this case, the time delay input to the block is

\[
t_i(t) = \frac{L}{v_i(t)}
\]
where $L$ is the length of the pipe and $v_i(t)$ is the speed of the fluid.

**Parameters**

**Select delay type — Type of delay**

Variable time delay | Variable transport delay

Specify the type of delay as Variable time delay or Variable transport delay.

The default value of this parameter depends on the block implementation: Variable time delay for the Variable Time Delay block, and Variable transport delay for the Variable Transport Delay block.

**Dependencies**

- Setting this parameter to Variable time delay enables the **Handle zero delay** parameter.
- Setting this parameter to Variable transport delay enables the **Absolute tolerance** and **State Name** parameters.

**Programmatic Use**

**Block Parameter:** VariableDelayType

Type: character vector, string

Values: 'Variable transport delay' | 'Variable time delay'

**Maximum delay — Maximum value of time delay input**

10 (default) | scalar | vector

Set the maximum value of the time delay input. This value defines the largest time delay input that this block allows. The block clips any delay that exceeds this value. This value cannot be negative. If the time delay becomes negative, the block clips it to zero and issues a warning message.

**Programmatic Use**

**Block Parameter:** MaximumDelay

Type: character vector, string

Value: scalar | vector

Default: '10'

**Initial output — Initial output**

0 (default) | scalar | vector
Specify the output that the block generates until the simulation time first exceeds the
time delay input.

**Dependencies**

- The initial output of this block cannot be inf or NaN.
- A Run-to-run tunable parameter cannot be changed during simulation run time.
  However, changing it before a simulation begins does not cause Accelerator or Rapid
  Accelerator to regenerate code.

**Programmatic Use**

**Block Parameter:** `InitialOutput`

**Type:** character vector, string

**Values:** scalar | vector

**Default:** '0'

**Initial buffer size — Initial memory allocation**

1024 (default) | scalar

Define the initial memory allocation for the number of input points to store. The input
points define the history of the input signal up to the current simulation time.

- If the number of input points exceeds the initial buffer size, the block allocates
  additional memory.
- After simulation ends, a message displays if the buffer is not sufficient and more
  memory must be allocated.

**Tips**

- Because allocating memory slows down simulation, choose this value carefully if
  simulation speed is an issue.
- For long time delays, this block might use a large amount of memory, particularly for
  dimensionalized input.

**Programmatic Use**

**Block Parameter:** `MaximumPoints`

**Type:** character vector, string

**Values:** scalar | vector

**Default:** '1024'

**Use fixed buffer size — Use fixed-size buffer**

off (default) | on
Selecting this check box uses a fixed-size buffer to save input data from previous time steps. When you clear this check box, the block does not use a fixed-size buffer.

The **Initial buffer size** parameter specifies the buffer size. If the buffer is full, new data replaces data already in the buffer. Simulink software uses linear extrapolation to estimate output values that are not in the buffer.

**Note** ERT or GRT code generation uses a fixed-size buffer even if you do not select this check box.

**Tips**

- If the input data is linear, selecting this check box can save memory.
- If the input data is nonlinear, do not select this check box. Doing so might yield inaccurate results.

**Programmatic Use**

**Block Parameter:** FixedBuffer  
**Type:** character vector, string  
**Values:** 'off' | 'on'  
**Default:** 'off'

**Handle zero delay — Use direct feedthrough**

*off* (default) | *on*

Selecting this check box converts this block to a direct feedthrough block. When you clear this check box, the block does not use direct feedthrough.

**Dependencies**

To enable this parameter, set **Select delay type** to **Variable time delay**.

**Programmatic Use**

**Block Parameter:** ZeroDelay  
**Type:** character vector, string  
**Values:** 'off' | 'on'  
**Default:** 'off'

**Direct feedthrough of input during linearization — Use direct feedthrough during linearization**

*off* (default) | *on*
When you select this parameter, the block outputs its input during linearization and trim, which sets the block mode to direct feedthrough. To disable direct feedthrough, clear this check box.

**Tips**

- Selecting this check box can cause a change in the ordering of states in the model when you use the functions `linmod`, `dlinmod`, or `trim`. To extract this new state ordering:

  1. Compile the model using the following command, where `model` is the name of the Simulink model.
     
     ```
     [sizes, x0, x_str] = model([],[],[],'lincompile');
     ```
  2. Terminate the compilation with the following command.
     
     ```
     model([],[],[],'term');
     ```

- The output argument `x_str`, which is a cell array of the states in the Simulink model, contains the new state ordering. When you pass a vector of states as input to the `linmod`, `dlinmod`, or `trim` functions, the state vector must use this new state ordering.

**Programmatic Use**

**Block Parameter:** TransDelayFeedthrough  
**Type:** character vector, string  
**Values:** 'off' | 'on'  
**Default:** 'off'

**Pade order (for linearization) — Order of Pade approximation**

θ (default) | scalar

Set the order of the Pade approximation for linearization routines.

- The default value is 0, which results in a unity gain with no dynamic states.
- Setting the order to a positive integer n adds n states to your model, but results in a more accurate linear model of the transport delay.

**Programmatic Use**

**Block Parameter:** PadeOrder  
**Type:** character vector, string  
**Values:** scalar  
**Default:** 'θ'
**Absolute tolerance — Absolute tolerance for computing block state**

auto (default) | positive, real, scalar or vector

Specify the absolute tolerance for computing the block state.

**Dependencies**

To enable this parameter, set **Select delay type** to Variable transport delay.

**Programmatic Use**

**Block Parameter:** AbsoluteTolerance  
**Type:** character vector, string  
**Values:** 'auto' | '-1' | any positive real scalar or vector  
**Default:** 'auto'

**State Name (e.g., 'position') — Unique name for each state**

' ' (default) | character vector, string

Assign a unique name to each state. If this field is blank, no name assignment occurs.

**Tips**

- To assign a name to a single state, enter the name between quotes, for example, 'velocity'.
- To assign names to multiple states, enter a comma-delimited list surrounded by braces, for example, {'a', 'b', 'c'}. Each name must be unique.
- The state names apply only to the selected block.
- The number of states must divide evenly among the number of state names.
- You can specify fewer names than states, but you cannot specify more names than states.

For example, you can specify two names in a system with four states. The first name applies to the first two states and the second name to the last two states.

- To assign state names with a variable in the MATLAB workspace, enter the variable without quotes. A variable can be a character vector, string, cell array, or structure.

**Dependencies**

To enable this parameter, set **Select delay type** to Variable transport delay.

**Programmatic Use**

**Block Parameter:** ContinuousStateAttributes
Type: character vector, string
Values: ' ' | user-defined character vector, user-defined string
Default: ' ' 

Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>double</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>yes</td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>no</td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
</tr>
</tbody>
</table>

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Not recommended for production-quality code. Relates to resource limits and restrictions on speed and memory often found in embedded systems. The code generated can contain dynamic allocation and freeing of memory, recursion, additional memory overhead, and widely-varying execution times. While the code is functionally valid and generally acceptable in resource-rich environments, smaller embedded targets often cannot support such code.

In general, consider using the Simulink Model Discretizer to map continuous blocks into discrete equivalents that support production code generation. To start the Model Discretizer, select Analysis > Control Design > Model Discretizer. One exception is the Second-Order Integrator block because, for this block, the Model Discretizer produces an approximate discretization.
See Also
Transport Delay | Variable Transport Delay

Topics
“State”

Introduced in R2007a
Variable Transport Delay

Delay input by variable amount of time

**Library:** Simulink / Continuous

## Description

The Variable Transport Delay and Variable Time Delay blocks appear as two blocks in the Simulink block library. However, they are the same Simulink block with different settings for the **Select delay type** parameter. Use this parameter to specify the mode in which the block operates.

### Variable Transport Delay

In this mode, the block output at the current time step is equal to the value of its data (top, or left) input at an earlier time step equal to the current time minus a transport delay.

\[
y(t) = u(t - t_d(t))
\]

Simulink software finds the transport delay, \( t_d(t) \), by solving the following equation:

\[
\int_{t - t_d(t)}^{t} \frac{1}{t_i(\tau)} d\tau = 1
\]

This equation involves an instantaneous time delay, \( t_i(t) \), given by the time delay (bottom, or right) input.

![Variable Transport Delay Block Diagram](image)
Suppose that you want to use this block to model the fluid flow through a pipe where the fluid speed varies with time. In this case, the time delay input to the block is

\[ t_i(t) = \frac{L}{v_i(t)} \]

where \( L \) is the length of the pipe and \( v_i(t) \) is the speed of the fluid.

**Variable Time Delay**

In this mode, the block has a data input, a time delay input, and a data output. (See “Port Location After Rotating or Flipping” for a description of the port order for various block orientations.) The output at the current time step equals the value of its data input at a previous time step. This time step is the current simulation time minus a delay time specified by the time delay input.

\[ y(t) = u(t - t_0) = u(t - \tau(t)) \]

During the simulation, the block stores time and input value pairs in an internal buffer. At the start of simulation, the block outputs the value of the Initial output parameter until the simulation time exceeds the time delay input. Then, at each simulation step, the block outputs the signal at the time that corresponds to the current simulation time minus the delay time.

If you want the output at a time between input storing times and the solver is a continuous solver, the block interpolates linearly between points. If the time delay is smaller than the step size, the block extrapolates an output point from a previous point. For example, consider a fixed-step simulation with a step size of 1 and the current time at \( t = 5 \). If the delay is 0.5, the block must generate a point at \( t = 4.5 \), but the most recent stored time value is at \( t = 4 \). Thus, the block extrapolates the input at 4.5 from the input at 4 and uses the extrapolated value as its output at \( t = 5 \).

Extrapolating forward from the previous time step can produce a less accurate result than extrapolating back from the current time step. However, the block cannot use the
current input to calculate its output value because the input port does not have direct feedthrough.

If the model specifies a discrete solver, the block does not interpolate between time steps. Instead, it returns the nearest stored value that precedes the required value.

**Ports**

**Input**

**Port_1 — Input signal**

scalar | vector | matrix

Input signal specified as a scalar, vector, or matrix.

Data Types: double

**t₀ — Time delay input**

scalar | vector | matrix

Time delay input specified as a scalar, vector, or matrix. When the block is in **Variable time delay** mode, this value specifies the time delay. For more information about that calculation, see “Variable Time Delay” on page 1-2264.

**Dependencies**

To enable this port, set **Select delay type** to **Variable time delay**.

Data Types: double

**t₁ — Instantaneous time delay input**

scalar | vector | matrix

Instantaneous time delay input specified as a scalar, vector, or matrix. When the block is in **Variable transport delay** mode, this value is used to calculate the transport delay. For more information about that calculation, see “Variable Transport Delay” on page 1-2263.

**Dependencies**

To enable this port, set **Select delay type** to **Variable transport delay**.
Data Types: double

Output

Port_1 — Delayed signal
scalar | vector | matrix

Output signal specified as a scalar, vector, or matrix.

Data Types: double

Parameters

Select delay type — Type of delay
Variable time delay | Variable transport delay

Specify the type of delay as Variable time delay or Variable transport delay.

The default value of this parameter depends on the block implementation: Variable time delay for the Variable Time Delay block, and Variable transport delay for the Variable Transport Delay block.

Dependencies

- Setting this parameter to Variable time delay enables the Handle zero delay parameter.
- Setting this parameter to Variable transport delay enables the Absolute tolerance and State Name parameters.

Programmatic Use

Block Parameter: VariableDelayType
Type: character vector, string
Values: 'Variable transport delay' | 'Variable time delay'

Maximum delay — Maximum value of time delay input
10 (default) | scalar | vector

Set the maximum value of the time delay input. This value defines the largest time delay input that this block allows. The block clips any delay that exceeds this value. This value cannot be negative. If the time delay becomes negative, the block clips it to zero and issues a warning message.
Programmatic Use

Block Parameter: MaximumDelay
Type: character vector, string
Value: scalar | vector
Default: '10'

Initial output — Initial output

0 (default) | scalar | vector

Specify the output that the block generates until the simulation time first exceeds the time delay input.

Dependencies

- The initial output of this block cannot be inf or NaN.
- A Run-to-run tunable parameter cannot be changed during simulation run time. However, changing it before a simulation begins does not cause Accelerator or Rapid Accelerator to regenerate code.

Programmatic Use

Block Parameter: InitialOutput
Type: character vector, string
Values: scalar | vector
Default: '0'

Initial buffer size — Initial memory allocation

1024 (default) | scalar

Define the initial memory allocation for the number of input points to store. The input points define the history of the input signal up to the current simulation time.

- If the number of input points exceeds the initial buffer size, the block allocates additional memory.
- After simulation ends, a message displays if the buffer is not sufficient and more memory must be allocated.

Tips

- Because allocating memory slows down simulation, choose this value carefully if simulation speed is an issue.
- For long time delays, this block might use a large amount of memory, particularly for dimensionalized input.
Programmatic Use
Block Parameter: MaximumPoints
Type: character vector, string
Values: scalar | vector
Default: '1024'

Use fixed buffer size — Use fixed-size buffer
off (default) | on

Selecting this check box uses a fixed-size buffer to save input data from previous time steps. When you clear this check box, the block does not use a fixed-size buffer.

The Initial buffer size parameter specifies the buffer size. If the buffer is full, new data replaces data already in the buffer. Simulink software uses linear extrapolation to estimate output values that are not in the buffer.

**Note** ERT or GRT code generation uses a fixed-size buffer even if you do not select this check box.

**Tips**

- If the input data is linear, selecting this check box can save memory.
- If the input data is nonlinear, do not select this check box. Doing so might yield inaccurate results.

Programmatic Use
Block Parameter: FixedBuffer
Type: character vector, string
Values: 'off' | 'on'
Default: 'off'

Handle zero delay — Use direct feedthrough
off (default) | on

Selecting this check box converts this block to a direct feedthrough block. When you clear this check box, the block does not use direct feedthrough.

**Dependencies**

To enable this parameter, set **Select delay type** to **Variable time delay**.
Programmatic Use

Block Parameter: ZeroDelay
Type: character vector, string
Values: 'off' | 'on'
Default: 'off'

Direct feedthrough of input during linearization — Use direct feedthrough during linearization
off (default) | on

When you select this parameter, the block outputs its input during linearization and trim, which sets the block mode to direct feedthrough. To disable direct feedthrough, clear this check box.

Tips

• Selecting this check box can cause a change in the ordering of states in the model when you use the functions linmod, dlinmod, or trim. To extract this new state ordering:

  1 Compile the model using the following command, where model is the name of the Simulink model.

     \[\text{[sizes, x0, x_str]} = \text{model([],[],[],'lincompile')}\];

  2 Terminate the compilation with the following command.

     \text{model([],[],[],'term')};

• The output argument x_str, which is a cell array of the states in the Simulink model, contains the new state ordering. When you pass a vector of states as input to the linmod, dlinmod, or trim functions, the state vector must use this new state ordering.

Programmatic Use

Block Parameter: TransDelayFeedthrough
Type: character vector, string
Values: 'off' | 'on'
Default: 'off'

Pade order (for linearization) — Order of Pade approximation
0 (default) | scalar

Set the order of the Pade approximation for linearization routines.
• The default value is 0, which results in a unity gain with no dynamic states.
• Setting the order to a positive integer \( n \) adds \( n \) states to your model, but results in a more accurate linear model of the transport delay.

**Programmatic Use**

**Block Parameter:** PadeOrder  
**Type:** character vector, string  
**Values:** scalar  
**Default:** '0'

**Absolute tolerance — Absolute tolerance for computing block state**

auto (default) | positive, real, scalar or vector

Specify the absolute tolerance for computing the block state.

**Dependencies**

To enable this parameter, set **Select delay type** to Variable transport delay.

**Programmatic Use**

**Block Parameter:** AbsoluteTolerance  
**Type:** character vector, string  
**Values:** 'auto' | '-1' | any positive real scalar or vector  
**Default:** 'auto'

**State Name (e.g., 'position') — Unique name for each state**

' ' (default) | character vector, string

Assign a unique name to each state. If this field is blank, no name assignment occurs.

**Tips**

• To assign a name to a single state, enter the name between quotes, for example, 'velocity'.
• To assign names to multiple states, enter a comma-delimited list surrounded by braces, for example, {'a', 'b', 'c'}. Each name must be unique.
• The state names apply only to the selected block.
• The number of states must divide evenly among the number of state names.
• You can specify fewer names than states, but you cannot specify more names than states.
For example, you can specify two names in a system with four states. The first name applies to the first two states and the second name to the last two states.

- To assign state names with a variable in the MATLAB workspace, enter the variable without quotes. A variable can be a character vector, string, cell array, or structure.

**Dependencies**

To enable this parameter, set **Select delay type** to **Variable transport delay**.

**Programmatic Use**

**Block Parameter:** ContinuousStateAttributes  
**Type:** character vector; string  
**Values:** ' ' | user-defined character vector, user-defined string  
**Default:** ' '  

**Block Characteristics**

<table>
<thead>
<tr>
<th>Data Types</th>
<th>double</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>yes</td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>no</td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
</tr>
</tbody>
</table>

**Extended Capabilities**

**C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

Not recommended for production-quality code. Relates to resource limits and restrictions on speed and memory often found in embedded systems. The code generated can contain
dynamic allocation and freeing of memory, recursion, additional memory overhead, and widely-varying execution times. While the code is functionally valid and generally acceptable in resource-rich environments, smaller embedded targets often cannot support such code.

In general, consider using the Simulink Model Discretizer to map continuous blocks into discrete equivalents that support production code generation. To start the Model Discretizer, select Analysis > Control Design > Model Discretizer. One exception is the Second-Order Integrator block because, for this block, the Model Discretizer produces an approximate discretization.

See Also
Entity Transport Delay | Transport Delay | Variable Time Delay

Topics
“State”

Introduced in R2007a
Variant Sink

Route amongst multiple outputs using Variants

Library: Simulink / Signal Routing

Description

The Variant Sink block has one input port and one or more output ports. You can define Variant choices as blocks that are connected to the output port so that, at most, one choice is active.

Each output port is associated with a Variant control. The Variant control that evaluates to `true`, determines which output port is active.

During simulation, Simulink connects the active choice directly to the input port of the Variant Sink block and ignores the inactive choices.

Ports

Input

Port_1 — Input signal

scalar | vector | matrix

Input signal, specified as a scalar, vector, or matrix, to be connected to the active output port.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `Boolean` | `fixed point` | `enumerated` | `bus` | `struct`
Output

**Port_1 — Output from first Variant**
scalar | vector | matrix

Output signal from the first Variant. The Variant control that evaluates to *true*, determines which output port is active.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus

**Port_N — Output from Nth Variant**
scalar | vector | matrix

Output signal from the Nth Variant. The Variant control that evaluates to *true*, determines which output port is active.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus

Parameters

**Variant control mode — Name of the Variant control mode**
Expression (default) | Label

To choose the active Variant based on the evaluation of the Variant conditions, use the Expression mode else select Label mode. When you select the **Variant control mode** as Label, the **Label mode active choice** option is available. In Label mode, Variant control need not be created in the global workspace. In Label mode, the Variant control is a string that is not evaluated and the choice used in simulation is determined by the **Label mode active choice** parameter. You can select an active Variant choice from **Label mode active choice** options.

When you select Label option, the Variant badge indicates the change.

**Note** When you promote **Label mode active choice** parameter to a mask, the **Variant control mode** is disabled.

- If the block is in Expression mode while promoting **Label mode active choice** parameter to mask, you can change the **Variant control mode** to Label by changing the promoted **Label mode active choice** parameter from the Mask dialog box.
- If the block is in Label mode while promoting Label mode active choice parameter to mask, you cannot change the Variant control mode to Expression mode.

For information about promoting parameters to mask, see “Promote Parameter to Mask”.

**Port — Number of connected input port**

No default

Number of the input port that is connected to one Variant choice upstream of the Variant Sink block. This value is read-only.

Click \[+\] to add a port or \[-\] to delete an existing one.

**Variant control expression — Variant controls available in the global workspace**

'Variant' (default) | boolean condition expression | Simulink.Variant object | Simulink.Parameter object | enum

Displays the Variant controls available in the global workspace. The Variant control can be a Boolean condition expression or a Simulink.Variant object representing a Boolean condition expression. If you want to generate code for your model, you must define the control variables as MATLAB variables.

To enter non-numeric Variant control values, use enumerated data. For information about using enumerated data, see “Use Enumerated Data in Simulink Models”

To enter a Variant name, double-click a **Variant control expression** cell in a new row and type in the Variant control expression. Click **Apply** after you edit a Variant control name. If you add or delete a Variant control without applying the changes, the previous edits on the Variant control name are lost.

**Programmatic Use**

**Block Parameter:** VariantControls

**Type:** cell array of character vectors

**Values:** Variant control that is associated with the Variant choice

**Default:** 'Variant'

**Condition (read-only) — Condition for Variant controls**

No default
Displays the **Condition** for the Variant controls that are Simulink.Variant objects. Create or change a Variant condition in the Simulink.Variant parameter dialog box or in the global workspace.

For more information, see “Create Variant Controls Programmatically” and Simulink.Variant.

**Label mode active choice — Name of Variant to use if Label control mode is selected**

Choice_1 (default) | name of variant control

When you select the **Variant control mode** as Label, the **Label mode active choice** option is available. You can select an active Variant choice from **Label mode active choice** options. You can also right-click the badge on the Variant Sink block and select **Label Mode Active Choice**.

The **Label mode active choice** drop-down list displays all Variant controls that are currently defined in the global workspace or a data dictionary. Use valid MATLAB identifiers to specify Variant controls. For more information, see Simulink.Variant.

**Note** **Label mode active choice** option is not available in Expression mode.

**Dependencies**

To enable this parameter, select Label mode.

**Programmatic Use**
**Block Parameter:** LabelModeActivechoice  
**Type:** character vector  
**Values:** Specified by the Variant control expression  
**Default:** Choice_1

**Allow zero active variant controls — Simulate model without active Variant choice**

off (default) | on

To simulate a model (containing a Variant block) without an active Variant choice, select the **Allow zero active variant controls** option. When this option is selected and there is no active Variant choice, Simulink disables all the blocks connected to the input and output stream of Variant Sink block. The removed blocks are ignored from update diagram or simulation.
If you do not select this option, Simulink generates an error when there is no active Variant choice.

When you select this option, the Variant badge indicates the change.

**Dependencies**

Expression option from **Variant control mode** is selected.

**Programmatic Use**

**Block Parameter:** AllowZeroVariantControls  
**Type:** character vector  
**Values:** 'off' | 'on'  
**Default:** 'off'

**Show variant condition on block — Annotate block ports**

off (default) | on

When you select this option, Simulink annotates each Variant control (condition expression) on the Variant Sink block ports.

**Programmatic Use**

**Block Parameter:** ShowConditionOnBlock  
**Type:** character vector  
**Values:** 'off' | 'on'  
**Default:** 'off'

**Analyze all choices during update diagram and generate preprocessor conditionals — Analyze all Variant choices during update diagram or simulation**

off (default) | on

When you select this option, Simulink analyzes all Variant choices during an update diagram or simulation. This analysis helps Simulink to maintain consistency of all Variant branches during simulation and code generation. Simulink routes the output of the active and inactive regions to an internal `VariantMerge` block.

When you select this option, the preprocessor conditionals (#if) are generated in the code with ERT-based targets.

When you select this option, the Variant badge indicates the change.

For more information, see “Represent Variant Source and Sink Blocks in Generated Code” (Embedded Coder)
Dependencies

• Expression option from Variant control mode is selected.
• The check box is available for generating ERT targets only.

Programmatic Use

Block Parameter: GeneratePreprocessorConditionals
Type: character vector
Values: 'off' | 'on'
Default: 'off'

Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>bus</th>
<th>double</th>
<th>enumerated</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
<th>string</th>
</tr>
</thead>
</table>

Direct Feedthrough  no
Multidimensional Signals  no
Variable-Size Signals  no
Zero-Crossing Detection  no

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.
See Also

Topics
“Introduction to Variant Controls”
“Define, Configure, and Activate Variants”
“Working with Variant Choices”
“Variant Systems” (Embedded Coder)
“Represent Variant Source and Sink Blocks in Generated Code” (Embedded Coder)
“Variants Example Models”

Introduced in R2016a
Variant Source

Route among multiple inputs using Variants

Library: Simulink / Signal Routing

Description

The Variant Source block has one or more input ports and one output port. You can define Variant choices as blocks that are connected to the input port so that, at most, one choice is active.

Each input port is associated with a Variant control. The Variant control that evaluates to true, determines which input port is active.

When the Analyze all choices during update diagram and generate preprocessor conditionals option in the block dialog box is cleared, then during simulation Simulink connects the active choice directly to the output port of the Variant Source block and ignores the inactive choices.

Ports

Input

Port_1 — Input port associated with first Variant control

scalar | vector | matrix

Input port associated with the first Variant control. The Variant control that evaluates to true, determines which input port is active.
Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus

Port_N — Input port associated with Nth Variant control
scalar | vector | matrix

Input port associated with the Nth Variant control. The Variant control that evaluates to true, determines which input port is active.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus

Output

Port_1 — Output of active Variant
scalar | vector | matrix

Output signal from the active Variant.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus

Parameters

Variant control mode — Name of the Variant control mode
Expression (default) | Label

To choose the active Variant based on the evaluation of the Variant conditions, use the Expression mode else select Label mode. When you select the Variant control mode as Label, the Label mode active choice option is available. In Label mode, Variant control need not be created in the global workspace. You can select an active Variant choice from Label mode active choice options.

When you select Label option, the Variant badge indicates the change.

Note When you promote Label mode active choice parameter to a mask, the Variant control mode is disabled.
• If the block is in Expression mode while promoting Label mode active choice parameter to mask, you can change the Variant control mode to Label by changing the promoted Label mode active choice parameter from the Mask dialog box.

• If the block is in Label mode while promoting Label mode active choice parameter to mask, you cannot change the Variant control mode to Expression mode.

For information about promoting parameters to mask, see “Promote Parameter to Mask”.

**Port — Number of connected input port**

*no default*

Number of the input port that is connected to one Variant choice upstream of the Variant Source block. This value is read-only.

Click ![+](image) to add a port or ![−](image) to delete an existing one.

**Variant control expression — Variant controls available in the global workspace**

'Variant' (default) | boolean condition expression | Simulink.Variant object | Simulink.Parameter object | enum

Displays the Variant controls available in the global workspace. The Variant control can be a Boolean condition expression or a Simulink.Variant object representing a Boolean condition expression. If you want to generate code for your model, you must define the control variables as Simulink.Parameter objects.

To enter non-numeric Variant control values, use enumerated data. For information about using enumerated data, see “Use Enumerated Data in Simulink Models”

To edit a Variant name, double-click a Variant control expression cell and type in the Variant control expression. Click **Apply** after you edit a Variant control name. If you add or delete a Variant control without applying the changes, the previous edits on the Variant control name are lost.

The Variant control that evaluates to true determines which input port must be active.

**Programmatic Use**

**Block Parameter:** VariantControls  
**Type:** cell array of character vectors  
**Value:** Variant control that is associated with the Variant choice
**Default:** 'Variant'

**Condition (read-only) — Condition for Variant controls**

no default

Displays the **Condition** for the Variant controls that are `Simulink.Variant` objects.
Create or change a Variant condition in the `Simulink.Variant` parameter dialog box or in the global workspace.

For more information, see “Create Variant Controls Programmatically” and `Simulink.Variant`.

**Label mode active choice — Name of Variant to use if Label control mode is selected**

Choice_1 (default) | name of Variant control

When you select the **Variant control mode** as Label, the **Label mode active choice** option is available. You can select an active Variant choice from **Label mode active choice** options. You can also right-click the badge on the Variant Source block and select **Label Mode Active Choice**.

The **Label mode active choice** drop-down list displays all Variant controls that are currently defined in the global workspace or a data dictionary. Use valid MATLAB identifiers to specify Variant controls. For more information, see `Simulink.Variant`.

**Note** **Label mode active choice** option is not available in **Expression mode**.

**Dependencies**

To enable this parameter, select **Label mode**.

**Programmatic Use**

**Block Parameter:** LabelModeActivechoice

**Type:** character vector

**Value:** Specified by the Variant control expression.

**Default:** Choice_1

**Allow zero active variant controls — Simulate model without active Variant choice**

off (default) | on
To simulate a model (containing a Variant block) without an active Variant choice, select the **Allow zero active variant controls** option. When this option is selected and there is no active Variant choice, Simulink disables all the blocks connected to the input and output stream of Variant Source block. The removed blocks are ignored from update diagram or simulation.

If you do not select this option, Simulink generates an error when there is no active Variant choice.

When you select this option, the Variant badge indicates the change.

**Dependencies**

Expression option from **Variant control mode** is selected.

**Programmatic Use**

**Block Parameter:** AllowZeroVariantControls  
**Type:** character vector  
**Values:** 'off' | 'on'  
**Default:** 'off'

**Show variant condition on block — Annotate block ports**  
off (default) | on

When you select this option, Simulink annotates each Variant control (condition expression) on the Variant Source block ports.

**Programmatic Use**

**Block Parameter:** ShowConditionOnBlock  
**Type:** character vector  
**Value:** 'off' | 'on'  
**Default:** 'off'

**Analyze all choices during update diagram and generate preprocessor conditionals — Analyze all Variant choices during update diagram or simulation**  
off (default) | on

When you select this option, Simulink analyzes all Variant choices during an update diagram or simulation. This analysis helps Simulink to maintain consistency of all Variant branches during simulation and code generation. Simulink routes the output of the active and inactive regions to an internal VariantMerge block.

When this option is selected, the preprocessor conditionals (#if) are generated in the code with ERT-based targets.
If this option is selected during code generation, the data type and the semantics at all input ports of the Variant Source block must be same to avoid failure.

When you select this option, the Variant badge indicates the change.

For more information, see “Represent Variant Source and Sink Blocks in Generated Code” (Embedded Coder)

**Dependencies**

- Expression option from **Variant control mode** is selected.
- The check box is available for generating ERT targets only.

**Programmatic Use**

**Block Parameter:** GeneratePreprocessorConditionals  
**Type:** character vector  
**Value:** 'off' | 'on'  
**Default:** 'off'

### Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>bus</th>
<th>double</th>
<th>enumerated</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
<th>string</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Direct Feedthrough</th>
<th>no</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multidimensional Signals</td>
<td>no</td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
</tr>
</tbody>
</table>
Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

See Also

Topics
“Introduction to Variant Controls”
“Define, Configure, and Activate Variants”
“Create Variant Controls Programmatically”
“Variant Systems” (Embedded Coder)
“Represent Variant Source and Sink Blocks in Generated Code” (Embedded Coder)
“Variants Example Models”

Introduced in R2016a
Variant Subsystem, Variant Model

Template subsystem containing Subsystem blocks or Model blocks as Variant choices

Library:
- Simulink / Ports & Subsystems
- HDL Coder / Ports & Subsystems

Description

The Variant Subsystem block can have at most one active choice for simulation. The Variant Subsystem block is a template preconfigured to contain two Subsystem blocks to use as Variant Subsystem choices.

A Variant Subsystem block can contain a mixture of Subsystem and Model blocks as Variant systems. This can also include Inport, Outport, and Connection Port blocks. There are no drawn connections inside the Variant Subsystem blocks.

A Variant Subsystem block with Model blocks as choices, is called Variant Model block.

Each Variant system is associated with a Variant control that is created in the global workspace. The Variant control determines which Variant system is active. The Variant control can be a condition expression, a Simulink.Variant object specifying a condition expression, or a default Variant. The Variant control that evaluates to true determines the active Variant.

When you select the Specify output when source is unconnected option in the Outport block that is in a Variant Subsystem block, you can specify a non-ground value as its output.

Note You must specify the correct data type in the Signal Attributes section of the Outport block dialog box.
Ports

During simulation, Simulink disables the inactive ports in a Variant Subsystem block.

Input

In_1 — Input port corresponding to root-level Inport blocks contained in Variant Subsystem
same data types accepted by Inport blocks

Each Subsystem or Model block contained within a Variant Subsystem represents one Variant system. If the inport names on a Variant system are a subset of the inport names used by the Variant Subsystem container block, then Variant system blocks can have different numbers of inports than the Variant Subsystem block has.

Output

Out_1 — Output port corresponding to root-level Outport blocks contained in Variant Subsystem
same data types accepted by Outport blocks

Each Subsystem or Model block contained within a Variant Subsystem represents one Variant system. If the outport names on a Variant system are a subset of the outport names used by the Variant Subsystem block, then Variant system blocks can have different numbers of outports than the Variant Subsystem block has.

Parameters

Variant control mode — Name of the Variant control mode
Expression (default) | Label

To choose the active Variant based on the evaluation of the Variant conditions, use the Expression mode else select Label mode. When you select the Variant control mode as Label, the Label mode active choice option is available. In Label mode, Variant control need not be created in the global workspace. You can select an active Variant choice from Label mode active choice options.

When you select Label mode, the Variant badge indicates the change.
**Note** When you promote the **Label mode active choice** parameter to a mask, the **Variant control mode** is disabled. While promoting **Label mode active choice** parameter to a mask in a nested model, ensure that you promote **Label mode active choice** parameter to the immediate parent Variant Subsystem block mask.

- If the block is in **Expression** mode while promoting **Label mode active choice** parameter to mask, you can change the **Variant control mode** to Label by changing the promoted **Label mode active choice** parameter from the Mask dialog box.
- If the block is in **Label** mode while promoting **Label mode active choice** parameter to mask, you cannot change the **Variant control mode** to **Expression** mode.

For information about promoting parameters to mask, see “Promote Parameter to Mask”.

**Variant choices (table of variant systems) — Table of variant choices, variant controls, and conditions**

<table>
<thead>
<tr>
<th>Empty table (default)</th>
</tr>
</thead>
</table>

The table has a row for each Variant system contained in the Variant Subsystem. If there are no Variant systems, the table is empty.

You can use buttons to the left of the **Variant choices** table to modify the elements in the table.

<table>
<thead>
<tr>
<th>To...</th>
<th>Click...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create and add a new subsystem choice: Place a new Subsystem Variant choice in the table and create a Subsystem block in the Variant Subsystem block diagram.</td>
<td><img src="image" alt="Create and add a new subsystem choice" /></td>
</tr>
<tr>
<td>Create and add a new model variant choice: Place a new Model Variant choice in the table and create a Model block in the Variant Subsystem block.</td>
<td><img src="image" alt="Create and add a new model variant choice" /></td>
</tr>
<tr>
<td>To...</td>
<td>Click...</td>
</tr>
<tr>
<td>----------------------------------------------------------------------</td>
<td>----------</td>
</tr>
<tr>
<td><strong>Create/Edit selected variant object:</strong> Create a Simulink.Variant object in the global workspace and open the Simulink.Variant object parameter dialog box to specify the Variant <strong>Condition</strong>.</td>
<td><img src="image" alt="Image" /></td>
</tr>
<tr>
<td><strong>Open selected variant choice block:</strong> Open the Subsystem block diagram for the selected row in the <strong>Variant choices</strong> table.</td>
<td><img src="image" alt="Image" /></td>
</tr>
<tr>
<td><strong>Refresh dialog information from Variant Subsystem contents:</strong> Update the <strong>Variant choices</strong> table according to the Variant system and values of the Variant control in the global workspace.</td>
<td><img src="image" alt="Image" /></td>
</tr>
</tbody>
</table>

**Name (read-only) — Variant system name**

' ' (default) | name of Subsystem or Model block contained in the Variant Subsystem

This read-only field is based on the Variant system name. To add a Subsystem Variant choice, click ![Image](image). To add a Model Variant choice, click ![Image](image).

**Variant control expression — Variant control in global workspace**

Variant (default) | boolean condition expression | a Simulink.Variant object representing a boolean condition expression | a Simulink.Parameter object (required for code generation) | enum

To enter a Variant name, double-click a **Variant control** cell in a new row and type in the **Variant control expression**.

To enter non-numeric Variant control values, use enumerated data. For information about using enumerated data, see “Use Enumerated Data in Simulink Models”

**Programmatic Use**

**Structure field:** Represented by the read-only `variant.Name` field in the Variant parameter structure

**Type:** character vector

**Value:** Variant control that is associated with the Variant choice

**Default:** 'variant'

**Condition (read-only) — Condition for Variant controls**

' ' (default)
This read-only field is based on the condition for the associated Variant control in the global workspace. Create or change a Variant condition in the Simulink.Variant parameter dialog box or in the global workspace.

**Label mode active choice — Name of Active choice if Label mode is selected**

Choice_1 (default) | on

When you select the **Variant control mode** to Label, the **Label mode active choice** option is available. You can select an active Variant choice from **Label mode active choice** options. You can also right-click the badge on the Variant Subsystem block and select **Label Mode Active Choice**.

For **Label mode active choice** option, the Variant control need not be a Boolean condition expression or a Simulink.Variant object. Variant controls that start with a % symbol are ignored.

**Note** **Label mode active choice** option is not available in Expression mode.

**Dependencies**

To enable this parameter, select Label option from **Variant control mode** parameter.

**Programmatic Use**

**Parameter:** LabelModeActivechoice  
**Type:** character vector  
**Value:** if no Label mode active choice is specified, the value is empty. If Label mode active choice is specified, the value is the name of the Label mode active choice.  
**Default:** ''

**Allow zero active variant controls — Simulate model without using active Variant**

off (default) | on

To simulate a model (containing a Variant system) without an active Variant choice, select the **Allow zero active variant controls** option. When you select this option and if there is no active Variant choice, Simulink disables all the blocks connected to the input and output stream of Variant Subsystem block. The disabled blocks are ignored from update diagram or simulation.

If you do not select this option, Simulink generates an error when there is no active Variant choice.
Dependencies

• The (default) option of Variant is not selected
• Expression option from **Variant control mode** is selected.

Programmatic Use
Parameter: **AllowZeroVariantControls**
Type: character vector
Value: 'off' | 'on'
Default: 'off'

Analyze all choices during update diagram and generate preprocessor conditionals — Generate preprocessor conditionals
off (default) | on

When generating code for an ERT target, this parameter determines whether Variant choices are enclosed within C preprocessor conditional statements (**#if**).

When you select this option, Simulink analyzes all Variant choices during an update diagram or simulation. This analysis provides early validation of the code generation readiness of all Variant choices.

When you select this option, the Variant badge changes.

Dependencies

• The check box is available for generating only ERT targets.
• Expression option from **Variant control mode** is selected.

Programmatic Use
Parameter: **GeneratePreprocessorConditionals**
Type: character vector
Value: 'off' | 'on'
Default: 'off'

Propagate conditions outside of variant subsystem — Propagate Variant conditions outside of Variant Subsystem block
off (default) | on
When you select this option, Simulink propagates the Variant conditions outside of the Variant Subsystem block to determine which components of the model are active during simulation.

When you select this option, the Variant badge indicates the change.

Programmatic Use
Parameter: PropagateVariantConditions
Type: character vector
Value: 'off' | 'on'
Default: 'off'

Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>bus</th>
<th>double</th>
<th>enumerated</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
<th>string</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>limited</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>limited</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

a. Actual data type or capability support depends on block implementation.
Extended Capabilities

**C/C++ Code Generation**
Generate C and C++ code using Simulink® Coder™.

Actual data type or capability support depends on block implementation.

**HDL Code Generation**
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic. Actual data type or capability support depends on block implementation.

**HDL Architecture**

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Module (default)</td>
<td>Generate code for the subsystem and the blocks within the subsystem. HDL Coder generates code for only the active variant.</td>
</tr>
<tr>
<td>BlackBox</td>
<td>Generate a black-box interface. That is, the generated HDL code includes only the input/output port definitions for the subsystem. In this way, you can use a subsystem in your model to generate an interface to existing manually written HDL code. The black-box interface generated for subsystems is similar to the interface generated for Model blocks, but without generation of clock signals.</td>
</tr>
<tr>
<td>No HDL</td>
<td>Remove the subsystem from the generated code. You can use the subsystem in simulation but treat it as a “no-op” in the HDL code.</td>
</tr>
</tbody>
</table>

**Black Box Interface Customization**

For the BlackBox architecture, you can customize port names and set attributes of the external component interface. See “Customize Black Box or HDL Cosimulation Interface” (HDL Coder).
## HDL Block Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>General</strong></td>
<td></td>
</tr>
<tr>
<td>AdaptivePipelining</td>
<td>Automatic pipeline insertion based on the synthesis tool, target frequency, and multiplier word-lengths. The default is inherit. See also “AdaptivePipelining” (HDL Coder).</td>
</tr>
<tr>
<td>BalanceDelays</td>
<td>Detects introduction of new delays along one path and inserts matching delays on the other paths. The default is inherit. See also “BalanceDelays” (HDL Coder).</td>
</tr>
<tr>
<td>ClockRatePipelining</td>
<td>Insert pipeline registers at a faster clock rate instead of the slower data rate. The default is inherit. See also “ClockRatePipelining” (HDL Coder).</td>
</tr>
<tr>
<td>ConstrainedOutputPipeline</td>
<td>Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. For more details, see “ConstrainedOutputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td>DistributedPipelining</td>
<td>Pipeline register distribution, or register retiming. The default is off. See also “DistributedPipelining” (HDL Coder).</td>
</tr>
<tr>
<td>DSPStyle</td>
<td>Synthesis attributes for multiplier mapping. The default is none. See also “DSPStyle” (HDL Coder).</td>
</tr>
<tr>
<td>FlattenHierarchy</td>
<td>Remove subsystem hierarchy from generated HDL code. The default is inherit. See also “FlattenHierarchy” (HDL Coder).</td>
</tr>
<tr>
<td>InputPipeline</td>
<td>Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “InputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td>OutputPipeline</td>
<td>Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “OutputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td>SharingFactor</td>
<td>Number of functionally equivalent resources to map to a single shared resource. The default is 0. See also “Resource Sharing” (HDL Coder).</td>
</tr>
</tbody>
</table>
**General**

| **StreamingFactor** | Number of parallel data paths, or vectors, that are time multiplexed to transform into serial, scalar data paths. The default is 0, which implements fully parallel data paths. See also “Streaming” (HDL Coder). |

**Target Specification**

This block cannot be the DUT, so the block property settings in the Target Specification tab are ignored.

**Restrictions**

- The DUT cannot be a Variant Subsystem.

**Fixed-Point Conversion**

Design and simulate fixed-point systems using Fixed-Point Designer™.

Actual data type or capability support depends on block implementation.

**See Also**

Inport | Outport | Simulink.Variant

**Topics**

“Variant Systems” (Embedded Coder)
“Define, Configure, and Activate Variants”
“Working with Variant Choices”
“Introduction to Variant Controls”
“Create Variant Controls Programmatically”

**Introduced in R2010b**
Vector Concatenate, Matrix Concatenate

Concatenate input signals of same data type to create contiguous output signal

**Library:**
- Simulink / Commonly Used Blocks
- Simulink / Math Operations
- Simulink / Signal Routing
- HDL Coder / Math Operations
- HDL Coder / Signal Routing

**Description**

The Concatenate block concatenates the input signals to create an output signal whose elements reside in contiguous locations in memory.

**Tip** The Concatenate block is useful for creating an output signal that is nonvirtual. However, to create a vector of function calls, use a Mux block instead.

You use a Concatenate block to define an array of buses. For details about defining an array of buses, see “Combine Buses into an Array of Buses”.

The Concatenate block operates in either vector or multidimensional array concatenation mode, depending on the setting of its **Mode** parameter. In either case, the block concatenates the inputs from the top to bottom, or left to right, input ports.

**Vector Mode**

In vector mode, all input signals must be either vectors or row vectors (1-by-M matrices) or column vectors (M-by-1 matrices) or a combination of vectors and either row or column vectors. When all inputs are vectors, the output is a vector.

If any of the inputs are row or column vectors, the output is a row or column vector, respectively.
**Multidimensional Array Mode**

Multidimensional array mode accepts vectors and arrays of any size. It assumes that the trailing dimensions are all ones for input signals with lower dimensionality. For example, if the output is 4-D and the input is \([2\times3]\) (2-D), this block treats the input as \([2\times3\times1\times1]\). The output is always an array. The **Concatenate dimension** parameter allows you to specify the output dimension along which the block concatenates its input arrays.

If you set the **Concatenate dimension** parameter to 2 and inputs are 2-D matrices, the block performs horizontal matrix concatenation and places the input matrices side-by-side to create the output matrix. For example, see the ex_concatenate_horizontal model:

If you set the **Concatenate dimension** parameter to 1 and inputs are 2-D matrices, the block performs vertical matrix concatenation and stacks the input matrices on top of each other to create the output matrix. For example, see the ex_concatenate_vertical model:

For horizontal concatenation, the input matrices must have the same column dimension. For vertical concatenation, the input matrices must have the same row dimension. All input signals must have the same dimension for all dimensions other than the concatenation dimensions.
If you set the **Mode** parameter to **Multidimensional array**, the **Concatenate dimension** parameter to 3, and the inputs are 2-D matrices, the block performs multidimensional matrix concatenation. For example, see the `ex_concatenate_multidims` model:

![Block diagram example](image)

**Ports**

**Input**

**Port_1 — First input to concatenate**

scalar | vector | matrix | N-D array

First input to concatenate, specified as a scalar, vector, matrix, or N-D array.

**Dependencies**

- Inputs must be of the same data type.
- Matrix and N-D array inputs are supported only when you set **Mode** to **Multidimensional array**.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus

**Port_N — Nth input to concatenate**

scalar | vector | matrix | N-D array

Nth input to concatenate, specified as a scalar, vector, matrix, or N-D array.
**Dependencies**

- To enable this port, set **Number of inputs** to an integer greater than or equal to 2.
- Inputs must be of the same data type.
- Matrix and N-D array inputs are supported only when you set **Mode** to **Multidimensional array**.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus

**Output**

**Port_1 — Concatenation of input signals**

Scalar | vector | matrix | N-D array

Concatenation of input signals, along specified dimension. Outputs have the same data type as the input.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated

**Parameters**

**Number of inputs — Number of input ports**

2 (default) | positive integer

Specify the number of inputs for the block as a real-valued, positive integer, less than or equal to 65536.

**Programmatic Use**

**Block Parameter:** NumInputs

**Type:** character vector

**Values:** positive integer

**Default:** '2'

**Mode — Type of concatenation**

Vector | Multidimensional array

Select the type of concatenation that this block performs. The default **Mode** of the Vector Concatenate block is Vector. The default **Mode** of the Matrix Concatenate block is Multidimensional array.
• When you select Vector the block performs vector concatenation (see “Vector Mode” on page 1-2297 for details).

• When you select Multidimensional array, the block performs matrix concatenation (see “Multidimensional Array Mode” on page 1-2298 for details).

**Programmatic Use**

**Block Parameter:** Mode

**Type:** character vector

**Values:** 'Vector' | 'Multidimensional array'

**Default:** 'Vector'

**Concatenate dimension — Output dimension along which to concatenate input arrays**

1 (default) | scalar integer

Specify the output dimension along which to concatenate the input arrays.

• To concatenate input arrays vertically, enter 1.
• To concatenate input arrays horizontally, enter 2.
• To perform multidimensional concatenation on the inputs, specify an integer greater than 2.

**Dependencies**

To enable this parameter, set Mode to Multidimensional array.

**Programmatic Use**

**Block Parameter:** ConcatenateDimension

**Type:** character vector

**Values:** scalar integer

**Default:** '1'

---

**Block Characteristics**

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>double</th>
<th>enumerated</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Direct Feedthrough</strong></td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Extended Capabilities

**C/C++ Code Generation**
Generate C and C++ code using Simulink® Coder™.

**HDL Code Generation**
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

**HDL Architecture**
This block has a single, default HDL architecture.

**HDL Block Properties**

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ConstrainedOutputPipeline</strong></td>
<td>Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. For more details, see “ConstrainedOutputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td><strong>InputPipeline</strong></td>
<td>Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “InputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td><strong>OutputPipeline</strong></td>
<td>Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “OutputPipeline” (HDL Coder).</td>
</tr>
</tbody>
</table>

Complex Data Support

This block supports code generation for complex signals.

Restrictions

HDL code generation does not support matrices at the input or output ports of the block.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

Design and simulate fixed-point systems using Fixed-Point Designer™.

See Also

Mux | cat

Topics

“Combine Buses into an Array of Buses”
“Creating, Concatenating, and Expanding Matrices” (MATLAB)

Introduced in R2009b
Weighted Sample Time

Support calculations involving sample time
Library: Simulink / Signal Attributes

Description

The Weighted Sample Time block outputs the weighted sample time or weighted sample rate. Because the Weighted Sample Time block is an implementation of the Weighted Sample Time Math, you can also add, subtract, multiply, or divide the input signal, u, by a weighted sample time, Ts. If the input signal is continuous, Ts is the sample time of the Simulink model. Otherwise, Ts is the sample time of the discrete input signal. If the input signal is constant, Simulink assigns a finite sample time to the block based on its connectivity and context.

You specify the math operation with the Operation parameter. The block can output just a weighted sample time (Ts Only) or a weighted sample rate (1/Ts Only).

Enter the weighting factor in the Weight value parameter. If the weight, w, is 1, that value does not appear in the equation on the block icon.

Tip You can use the Weighted Sample Time and Weighted Sample Time Math blocks to extract the sample time from a Simulink signal. To do so, set the Operation parameter to Ts and the Weight value to 1.0. In this configuration, the block outputs the sample time of the input signal.

The block computes its output using the precedence rules for MATLAB operators. See “Operator Precedence” (MATLAB) for information. For example, if the Operation parameter specifies +, the block calculates output using this equation:

\[ u + (Ts \times w) \]

However, if the Operation parameter specifies /, the block calculates output using this equation:
\[
\frac{(u / Ts)}{w}
\]

**Ports**

**Input**

**Port_1 — Input signal**
scalar | vector | matrix

Specify the input signal, \( u \), as a scalar, vector, or matrix. Depending on the value of the **Operation** parameter, the block can add, subtract, multiply, or divide the input signal by weighted sample time or just output the weighted sample time or weighted sample rate.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point

**Output**

**Port_1 — Output signal**
scalar | vector | matrix

Output the weighted sample time or sample rate of the input signal, or output the input signal adjusted by the weighted sample time, \( Ts \). If the input signal is continuous, \( Ts \) is the sample time of the Simulink model. Otherwise, \( Ts \) is the sample time of the discrete input signal. When the input signal is constant, Simulink assigns a finite sample time to the block based on its connectivity and context.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point

**Parameters**

**Main**

**Operation — Math operation**
Ts Only (default) | + | - | * | / | 1/Ts Only

1-2305
Specify the operation to use for adjusting the input signal. You can select: +, -, *, /, Ts Only, or 1/Ts Only.

**Programmatic Use**  
**Block Parameter:** TsamMathOp  
**Type:** character vector  
**Values:** '+', '-' | '*' | '/' | 'Ts Only' | '1/Ts Only'  
**Default:** 'Ts Only'

**Weight value — Weight of sample time**  
1.0 (default) | real-valued scalar

Enter the weight of the sample time as a real-valued scalar.

**Programmatic Use**  
**Block Parameter:** weightValue  
**Type:** character vector  
**Values:** real-valued scalar  
**Default:** '1.0'

**Implement using — Method for adjusting sample time**  
**Online Calculations** (default) | **Offline Scaling Adjustment**

Select one of two modes: online calculations or offline scaling adjustment.

<table>
<thead>
<tr>
<th>Result of (Ts * w)</th>
<th>Output Data Type of Two Modes</th>
<th>Block Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>A power of 2, or an integer value</td>
<td>The same, when <strong>Output data type</strong> is Inherit: Inherit via internal rule</td>
<td>Equally efficient in both modes</td>
</tr>
<tr>
<td>Not power of 2 and not an integer value</td>
<td>Different</td>
<td>More efficient for the offline scaling mode</td>
</tr>
</tbody>
</table>

**Note** When the **Implement using** parameter is not visible, operations default to online calculations.

**Dependencies**

To enable this parameter, set **Operation** to * or /.
Programmatic Use
Block Parameter: TsampMathImp
Type: character vector
Values: 'Online Calculations' | 'Offline Scaling Adjustment'
Default: 'Online Calculations'

Signal Attributes

Output data type — Data type of output signal
Inherit: Inherit via internal rule (default)| Inherit: Inherit via back propagation | <data type expression>

Specify the data type for the output.

Programmatic Use
Block Parameter: OutDataTypeStr
Type: character vector
Values: 'Inherit: Inherit via internal rule' | 'Inherit: Inherit via back propagation' | '<data type expression>'
Default: 'Inherit: Inherit via internal rule'

Integer rounding mode — Rounding mode for fixed-point operations
Floor (default) | Ceiling | Convergent | Nearest | Round | Simplest | Zero

Specify the rounding mode for fixed-point operations. For more information, see “Rounding” (Fixed-Point Designer).

Block parameters always round to the nearest representable value. To control the rounding of a block parameter, enter an expression using a MATLAB rounding function into the mask field.

Dependencies

To enable this parameter, set Operation to +, -, *, or /. If you set the Operation parameter to * or /, you must also set Implement using to Online Calculations.

Programmatic Use
Block Parameter: RndMeth
Type: character vector
Values: 'Ceiling' | 'Convergent' | 'Floor' | 'Nearest' | 'Round' | 'Simplest' | 'Zero'
Default: 'Floor'
**Saturate on integer overflow — Method of overflow action**

*off (default) | on*

Specify whether overflows saturate or wrap.

<table>
<thead>
<tr>
<th>Action</th>
<th>Rationale</th>
<th>Impact on Overflows</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Select this check box (on).</td>
<td>Your model has possible overflow, and you want explicit saturation protection in the generated code.</td>
<td>Overflows saturate to either the minimum or maximum value that the data type can represent.</td>
<td>The maximum value that the int8 (signed, 8-bit integer) data type can represent is 127. Any block operation result greater than this maximum value causes overflow of the 8-bit integer. With the check box selected, the block output saturates at 127. Similarly, the block output saturates at a minimum output value of -128.</td>
</tr>
<tr>
<td>Do not select this check box (off).</td>
<td>You want to optimize efficiency of your generated code.</td>
<td>Overflows wrap to the appropriate value that is representable by the data type.</td>
<td>The maximum value that the int8 (signed, 8-bit integer) data type can represent is 127. Any block operation result greater than this maximum value causes overflow of the 8-bit integer. With the check box cleared, the software interprets the overflow-causing value as int8, which can produce an unintended result. For example, a block result of 130 (binary 1000 0010) expressed as int8, is -126.</td>
</tr>
</tbody>
</table>

When you select this check box, saturation applies to every internal operation on the block, not just the output, or result. Usually, the code generation process can detect when
overflow is not possible. In this case, the code generator does not produce saturation code.

Dependencies

To enable this parameter, set **Operation** to +, -, *, or / . If you set the **Operation** parameter to * or / , you must also set **Implement using** to Online Calculations.

**Programmatic Use**

**Block Parameter:** SaturateOnIntegerOverflow

**Type:** character vector

**Values:** 'off' | 'on'

**Default:** 'off'

**Block Characteristics**

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>double</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Extended Capabilities**

**C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

**Fixed-Point Conversion**

Design and simulate fixed-point systems using Fixed-Point Designer™.
See Also
Probe | Weighted Sample Time Math

Topics
“What Is Sample Time?”
“Specify Sample Time”
“View Sample Time Information”

Introduced before R2006a
Waveform Generator

Output waveforms using signal notations
Library: Simulink / Sources

Description

The Waveform Generator block outputs waveforms based on signal notations that you enter in the Waveform Definition table.

This block supports these syntaxes for the signal notations:

- Function syntax — Specify all arguments in the specific order for the signal syntax (see “Algorithms” on page 1-2320).
- Name-value syntax — Specify optional comma-separated pairs of Name,Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside single quotes (‘ ’). You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN. For more information, see “Algorithms” on page 1-2320.

This block supports normal, accelerator, and rapid accelerator modes and fast restart.

Supported Operators

<table>
<thead>
<tr>
<th>Operation</th>
<th>Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>Absolute value</td>
<td>abs()</td>
</tr>
<tr>
<td>Addition</td>
<td>+</td>
</tr>
<tr>
<td>Division</td>
<td>/</td>
</tr>
<tr>
<td>Multiplication</td>
<td>*</td>
</tr>
<tr>
<td>Parentheses</td>
<td>()</td>
</tr>
<tr>
<td>Subtraction</td>
<td>-</td>
</tr>
<tr>
<td>Unary minus</td>
<td>-</td>
</tr>
</tbody>
</table>
The Waveform block observes the following rules of operator precedence:

1  (  )
2  + -  (unary)
3  * /
4  + -

**Supported Operations**

The Waveform Generator block outputs one signal at a time. You can change this output signal. Express frequency and phase offset parameters in radians. You can also:

- Nest signal notations, for example:

  \[
  \text{sin('Amplitude',sin('Amplitude',1,'Frequency',1,'Phase',0),'Frequency',1,'Phase',1)}
  \]
Reference real scalar variables in the base or model workspace, for example:

\[
\text{sin('Amplitude',x,'Frequency',y,'Phase',z)}
\]

\(x, y,\) and \(z\) exist in the base workspace.

For more information on waveforms, see the Algorithms section.

To quickly determine the response of a system to different types of inputs, you can vary the output signal of the Waveform Generator block while a simulation is in progress.

**Limitations**

- You cannot tune the parameters of a waveform, such as frequency or amplitude, during execution of the code that you generate by using Simulink Coder. Instead, you
can generate code that enables you to switch between waveform variants that you specify. For more information, see “Switch Between Output Waveforms During Code Execution for Waveform Generator Block” (Simulink Coder).

**Ports**

**Output**

*Port_1 — Generated output signal*

scalar | vector

Output signal specified by an entry in the **Waveform Definition** table.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | bus

**Parameters**

**Main**

*Output Signal — Waveform for output signal*

1 (default) | integer

Select waveform definition to specify the output signal. The number corresponds to the line item in the **Waveform Definition** table. You can change this parameter while a simulation is running.

**Programmatic Use**

**Block Parameter:** SelectedSignal

**Type:** character vector

**Values:** scalar

**Default:** '1'

*Waveform Definition — Waveform signal notations*

custom | gaussian(mean, variance, seed) | pulse(amplitude, trigger_time, duration) | sawtooth(amplitude, frequence, phase_offset) | sin(amplitude, frequence, phase_offset) |
Enter signal notations in the **Waveform Definition** table, one waveform definition per line. For syntax details, see Algorithms.

### Signal Attributes

Click the **Show data type assistant** button to display the **Data Type Assistant**, which helps you set the data type attributes. For more information, see “Specify Data Types Using Data Type Assistant”.

**Output minimum — Minimum output value for range checking**

[] (default) | scalar

Lower value of the output range that Simulink checks.

Simulink uses the minimum to perform:

- Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters”) for some blocks.
- Simulation range checking (see “Signal Ranges” and “Enable Simulation Range Checking”).
- Automatic scaling of fixed-point data types.
- Optimization of the code that you generate from the model. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. For more information, see “Optimize using the specified minimum and maximum values” (Simulink Coder).

**Note** **Output minimum** does not saturate or clip the actual output signal. Use the Saturation block instead.

**Programmatic Use**

**Block Parameter**: OutMin  
**Type**: character vector  
**Values**: '[]'| scalar  
**Default**: '[]'
Output maximum — Maximum output value for range checking

[ ] (default) | scalar

Upper value of the output range that Simulink checks.

Simulink uses the maximum value to perform:

- Parameter range checking (see “Specify Minimum and Maximum Values for Block Parameters”) for some blocks.
- Simulation range checking (see “Signal Ranges” and “Enable Simulation Range Checking”).
- Automatic scaling of fixed-point data types.
- Optimization of the code that you generate from the model. This optimization can remove algorithmic code and affect the results of some simulation modes such as SIL or external mode. For more information, see “Optimize using the specified minimum and maximum values” (Simulink Coder).

**Note** Output maximum does not saturate or clip the actual output signal. Use the Saturation block instead.

**Programmatic Use**

**Block Parameter:** OutMax

**Type:** character vector

**Values:** ['[ ]'] | scalar

**Default:** ['[ ]']

Output data type — Specify the output data type

double (default) | Inherit: Inherit via back propagation | single | int8 | int16 | int32 | uint8 | uint16 | uint32 | int64 | uint64 | fixdt(1,16,2^0,0) | <data type expression> | ...

Choose the data type for the output. The type can be inherited, specified directly, or expressed as a data type object such as Simulink.NumericType.

**Programmatic Use**

**Block Parameter:** OutDataTypeStr

**Type:** character vector

**Values:** Inherit: Inherit via back propagation, single, int8, int16, uint8, uint16, int32, uint32, int64, uint64,
fixdt(1,16,0), fixdt(1,16,2^0,0), fixdt(1,16,2^0,0). '<data type expression>'
Default: 'Double'

Lock output data type setting against changes by the fixed-point tools — Prevent fixed-point tools from overriding data types
off (default) | on

Select to lock the output data type setting of this block against changes by the Fixed-Point Tool and the Fixed-Point Advisor. For more information, see “Use Lock Output Data Type Setting” (Fixed-Point Designer).

Programmatic Use
Block Parameter: LockScale
Type: character vector
Values: 'off' | 'on'
Default: 'off'

Lock data type settings against changes by the fixed-point tools — Prevent fixed-point tools from overriding data types
off (default) | on

Select to lock data type settings of this block against changes by the Fixed-Point Tool and the Fixed-Point Advisor. For more information, see “Lock the Output Data Type Setting” (Fixed-Point Designer).

Programmatic Use
Block Parameter: LockScale
Values: 'off' | 'on'
Default: 'off'

Integer rounding mode — Specify the rounding mode for fixed-point operations
Floor (default) | Ceiling | Convergent | Nearest | Round | Simplest | Zero

Choose one of these rounding modes.

Ceiling
Rounds both positive and negative numbers toward positive infinity. Equivalent to the MATLAB ceil function.

Convergent
Rounds number to the nearest representable value. If a tie occurs, rounds to the nearest even integer. Equivalent to the Fixed-Point Designer convergent function.
Floor
 ROUNDS both positive and negative numbers toward negative infinity. Equivalent to the MATLAB floor function.

Nearest
 ROUNDS number to the nearest representable value. If a tie occurs, rounds toward positive infinity. Equivalent to the Fixed-Point Designer nearest function.

Round
 ROUNDS number to the nearest representable value. If a tie occurs, rounds positive numbers toward positive infinity and rounds negative numbers toward negative infinity. Equivalent to the Fixed-Point Designer round function.

Simplest
 AUTOMATICALLY chooses between round toward floor and round toward zero to generate rounding code that is as efficient as possible.

Zero
 ROUNDS number toward zero. Equivalent to the MATLAB fix function.

Programmatic Use
Block Parameter: RndMeth
Type: character vector
Values: 'Ceiling' | 'Convergent' | 'Floor' | 'Nearest' | 'Round' | 'Simplest' | 'Zero'
Default: 'Floor'

See Also

For more information, see “Rounding” (Fixed-Point Designer).

Saturate on integer overflow — Method of overflow action
off (default) | on

Specify whether overflows saturate or wrap.
<table>
<thead>
<tr>
<th>Action</th>
<th>Rationale</th>
<th>Impact on Overflows</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Select this check box (on).</td>
<td>Your model has possible overflow, and you want explicit saturation protection in the generated code.</td>
<td>Overflows saturate to either the minimum or maximum value that the data type can represent.</td>
<td>The maximum value that the int8 (signed, 8-bit integer) data type can represent is 127. Any block operation result greater than this maximum value causes overflow of the 8-bit integer. With the check box selected, the block output saturates at 127. Similarly, the block output saturates at a minimum output value of -128.</td>
</tr>
<tr>
<td>Do not select this check box (off).</td>
<td>You want to optimize efficiency of your generated code. You want to avoid overspecifying how a block handles out-of-range signals. For more information, see “Check for Signal Range Errors”.</td>
<td>Overflows wrap to the appropriate value that is representable by the data type.</td>
<td>The maximum value that the int8 (signed, 8-bit integer) data type can represent is 127. Any block operation result greater than this maximum value causes overflow of the 8-bit integer. With the check box cleared, the software interprets the overflow-causing value as int8, which can produce an unintended result. For example, a block result of 130 (binary 1000 0010) expressed as int8, is -126.</td>
</tr>
</tbody>
</table>

When you select this check box, saturation applies to every internal operation on the block, not just the output, or result. Usually, the code generation process can detect when overflow is not possible. In this case, the code generator does not produce saturation code.
Programmatic Use

Block Parameter: `SaturateOnIntegerOverflow`
- **Type**: character vector
- **Values**: 'off' | 'on'
- **Default**: 'off'

Sample time — Time interval between samples

Specify the time interval between samples. To inherit the sample time, set this parameter to `-1`. See “Specify Sample Time” for more information.

Programmatic Use

Block Parameter: `SampleTime`
- **Type**: character vector
- **Values**: scalar
- **Default**: '0.1'

Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>double</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Algorithms

Enter signal notations in the **Waveform Definition** table, one waveform definition per line. To add a waveform definition, click **Add**. The new waveform appears as an empty character vector. The block interprets empty character vectors or white space character vectors as ground.
To remove a waveform definition, click **Remove**. You can select multiple waveforms using **Ctrl+click** or **Shift+click**.

**Constant**

Constant values can be:

- Numbers
- Workspace variables
  - Scalar, real variables only
  - Built-in MATLAB constant, \( \pi \)

**Examples**

- 1
- 1.1
- \( x \)
- \( \pi \)

**Gaussian Noise**

**Syntax**

\[
\text{gaussian}(\text{mean}, \text{variance}, \text{seed})
\]

\[
\text{gaussian}('\text{Mean}', \text{mean}, '\text{Variance}', \text{variance}, '\text{Seed}', \text{seed})
\]

**Input Arguments**

- **mean** — Mean value of the random variable output.
  - Default: 0
- **variance** — Standard deviation squared of the random variable output.
  - Default: 1
- **Value**: Positive constant or positive real scalar variable
- **seed** — Initial seed value for the random number generator.
• Default: 0
• Value: Constant or real scalar variable

Example

`gaussian('Mean',0,'Variance',10,'Seed',1)`

---

**Pulse**

**Syntax**

`pulse(amplitude,trigger_time,duration)`

`pulse('Amplitude',amplitude,'TriggerTime',trigger_time,'Duration',duration)`

**Input Arguments**

• `amplitude` — Value of signal when pulse is high.
  • Default: 1
• `trigger_time` — Elapsed simulation time when signal changes to amplitude, in seconds.
• Default: 1
• Value: Constant or real scalar variable
• duration — How long the signal remains at the given amplitude before returning to ground, in seconds.
  • Default: 1
  • Value: Positive constant or positive real scalar variable

Example

```matlab
pulse('Amplitude',1,'TriggerTime',1,'Duration',1)
```

![Graph showing a pulse wave form]

### Sawtooth

**Syntax**

```matlab
sawtooth(amplitude,frequency,phase_offset)
sawtooth('Amplitude',amplitude,'Frequency',frequency,'Phase',phase_offset)
```
### Input Arguments

- **amplitude** — Sawtooth peak value.
  - Default: 1
- **frequency** — Waveform frequency, in rad/s.
  - Default: 1
- **phase_offset** — Horizontal signal shift, based on elapsed simulation time, in seconds.
  - Default: 0

### Example

```
sawtooth('Amplitude',1,'Frequency',1,'Phase',0)
```
Sine Wave

Syntax

\[
sin(\text{amplitude}, \text{frequency}, \text{phase}\_\text{offset})
\]

\[
sin(\text{'Amplitude'}, \text{amplitude}, \text{'Frequency'}, \text{frequency}, \text{'Phase'}, \text{phase}\_\text{offset})
\]

Input Arguments

- `amplitude` — Amplitude of sine wave.
  - Default: 1
- `frequency` — Waveform frequency, in rad/s.
  - Default: 1
- `phase_offset` — Phase offset, in rads.
  - Default: 0

Examples

\[
sin(\text{'Amplitude'},1,\text{'Frequency'},1,\text{'Phase'},0)
\]

To get the cosine waveform:
sin('Amplitude',1,'Frequency',1,'Phase',pi/2)

Square

Syntax

square(amplitude,frequency,phase_delay,duty_cycle)
square('Amplitude',amplitude,'Frequency',frequency,'Phase',phase_delay,...
'DutyCycle',duty_cycle)

Input Arguments

• amplitude — Amplitude of signal.
  • Default: 1
• frequency — Waveform frequency in rad/s.
  • Default: 1
• phase_delay — Horizontal signal shift based on elapsed simulation time, in seconds.
  • Default: 0
• duty_cycle — Percentage of high signal per period (0-100%). The block clips the minimum signal to 0% and the maximum signal to 100%.
• Default: 50

**Example**

```matlab
square('Amplitude',1,'Frequency',1,'Phase',0,'DutyCycle',50)
```

---

**Step**

**Syntax**

```matlab
step(step_time,initial_value,final_value)
step('StepTime',step_time,'InitialValue',initial_value,'FinalValue',final_value)
```

**Input Arguments**

- **step_time** — Elapsed simulation time when signal changes from **initial value** to **final value**, in seconds.
  
  - Default: 1
  
  - Value: Constant or positive real scalar variable.

- **initial_value** — Value of signal when elapsed simulation time is less than **step_time**, in seconds.
• Default: 0
• `final_value` — Value of signal when elapsed simulation time is greater than or equal to step time, in seconds.
  • Default: 1

**Example**

```matlab
step('StepTime',1,'InitialValue',0,'FinalValue',1)
```

---

**Extended Capabilities**

**C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

Not recommended for production code.
Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

See Also
Repeating Sequence | Signal Editor

Topics
“Signal Basics”

Introduced in R2015b
Weighted Sample Time Math

Support calculations involving sample time

Library: Simulink / Math Operations

Description

The Weighted Sample Time Math block adds, subtracts, multiplies, or divides its input signal, \( u \), by a weighted sample time, \( Ts \). If the input signal is continuous, \( Ts \) is the sample time of the Simulink model. Otherwise, \( Ts \) is the sample time of the discrete input signal. If the input signal is constant, Simulink assigns a finite sample time to the block based on its connectivity and context.

You specify the math operation with the **Operation** parameter. The block can output just a weighted sample time (**Ts Only**) or a weighted sample rate (**1/Ts Only**).

Enter the weighting factor in the **Weight value** parameter. If the weight, \( w \), is 1, that value does not appear in the equation on the block icon.

**Tip** You can use the Weighted Sample Time and Weighted Sample Time Math blocks to extract the sample time from a Simulink signal. To do so, set the **Operation** parameter to **Ts** and the **Weight value** to **1.0**. In this configuration, the block outputs the sample time of the input signal.

The block computes its output using the precedence rules for MATLAB operators. See “Operator Precedence” (MATLAB) for information. For example, if the **Operation** parameter specifies +, the block calculates output using this equation:

\[ u + (Ts \times w) \]

However, if the **Operation** parameter specifies /, the block calculates output using this equation:

\[ (u / Ts) / w \]
Ports

Input

Port_1 — Input signal
scalar | vector | matrix

Specify the input signal, \( u \), as a scalar, vector, or matrix. Depending on the value of the Operation parameter, the block can add, subtract, multiply, or divide the input signal by weighted sample time or just output the weighted sample time or weighted sample rate.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point

Output

Port_1 — Output signal
scalar | vector | matrix

Output the weighted sample time or sample rate of the input signal, or output the input signal adjusted by the weighted sample time, \( Ts \). If the input signal is continuous, \( Ts \) is the sample time of the Simulink model. Otherwise, \( Ts \) is the sample time of the discrete input signal. When the input signal is constant, Simulink assigns a finite sample time to the block based on its connectivity and context.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point

Parameters

Main

Operation — Math operation
+ (default) | - | * | / | Ts Only | 1/Ts Only

Specify the operation to use for adjusting the input signal. You can select: +, -, *, /, Ts Only, or 1/Ts Only.
Programmatic Use
Block Parameter: TsamMathOp
Type: character vector
Values: '"+' | '-' | '*' | '/' | 'Ts Only' | '1/Ts Only'
Default: '"+'

Weight value — Weight of sample time
1.0 (default) | real-valued scalar

Enter the weight of the sample time as a real-valued scalar.

Programmatic Use
Block Parameter: weightValue
Type: character vector
Values: real-valued scalar
Default: '"1.0'"'

Implement using — Method for adjusting sample time
Online Calculations (default) | Offline Scaling Adjustment

Select one of two modes: online calculations or offline scaling adjustment.

<table>
<thead>
<tr>
<th>Result of (Ts * w)</th>
<th>Output Data Type of Two Modes</th>
<th>Block Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>A power of 2, or an integer value</td>
<td>The same, when Output data type is Inherit: Inherit via internal rule</td>
<td>Equally efficient in both modes</td>
</tr>
<tr>
<td>Not power of 2 and not an integer value</td>
<td>Different</td>
<td>More efficient for the offline scaling mode</td>
</tr>
</tbody>
</table>

**Note** When the Implement using parameter is not visible, operations default to online calculations.

**Dependencies**
To enable this parameter, set Operation to * or /. 
Programmatic Use

Block Parameter: TsampMathImp
Type: character vector
Values: 'Online Calculations' | 'Offline Scaling Adjustment'
Default: 'Online Calculations'

Signal Attributes

Output data type — Data type of output signal
Inherit: Inherit via internal rule (default) | Inherit: Inherit via back propagation | <data type expression>

Specify the data type for the output.

Programmatic Use

Block Parameter: OutDataTypeStr
Type: character vector
Values: 'Inherit: Inherit via internal rule' | 'Inherit: Inherit via back propagation' | '<data type expression>'
Default: 'Inherit: Inherit via internal rule'

Integer rounding mode — Rounding mode for fixed-point operations
Floor (default) | Ceiling | Convergent | Nearest | Round | Simplest | Zero

Specify the rounding mode for fixed-point operations. For more information, see “Rounding” (Fixed-Point Designer).

Block parameters always round to the nearest representable value. To control the rounding of a block parameter, enter an expression using a MATLAB rounding function into the mask field.

Dependencies

To enable this parameter, set Operation to +, -, *, or /. If you set the Operation parameter to * or /, you must also set Implement using to Online Calculations.

Programmatic Use

Block Parameter: RndMeth
Type: character vector
Values: 'Ceiling' | 'Convergent' | 'Floor' | 'Nearest' | 'Round' | 'Simplest' | 'Zero'
Default: 'Floor'
**Saturate on integer overflow — Method of overflow action**

*off (default) | on*

Specify whether overflows saturate or wrap.

<table>
<thead>
<tr>
<th>Action</th>
<th>Rationale</th>
<th>Impact on Overflows</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Select this check box (on).</td>
<td>Your model has possible overflow, and you want explicit saturation protection in the generated code.</td>
<td>Overflows saturate to either the minimum or maximum value that the data type can represent.</td>
<td>The maximum value that the int8 (signed, 8-bit integer) data type can represent is 127. Any block operation result greater than this maximum value causes overflow of the 8-bit integer. With the check box selected, the block output saturates at 127. Similarly, the block output saturates at a minimum output value of -128.</td>
</tr>
<tr>
<td>Do not select this check box (off).</td>
<td>You want to optimize efficiency of your generated code. You want to avoid overspecifying how a block handles out-of-range signals. For more information, see “Check for Signal Range Errors”.</td>
<td>Overflows wrap to the appropriate value that is representable by the data type.</td>
<td>The maximum value that the int8 (signed, 8-bit integer) data type can represent is 127. Any block operation result greater than this maximum value causes overflow of the 8-bit integer. With the check box cleared, the software interprets the overflow-causing value as int8, which can produce an unintended result. For example, a block result of 130 (binary 1000 0010) expressed as int8, is -126.</td>
</tr>
</tbody>
</table>

When you select this check box, saturation applies to every internal operation on the block, not just the output, or result. Usually, the code generation process can detect when
overflow is not possible. In this case, the code generator does not produce saturation code.

**Dependencies**

To enable this parameter, set `Operation` to `+`, `-`, `*`, or `/`. If you set the `Operation` parameter to `*` or `/`, you must also set `Implement using` to `Online Calculations`.

**Programmatic Use**

**Block Parameter:** SaturateOnIntegerOverflow  
**Type:** character vector  
**Values:** 'off' | 'on'  
**Default:** 'off'

**Block Characteristics**

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>double</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Extended Capabilities**

**C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

**Fixed-Point Conversion**

Design and simulate fixed-point systems using Fixed-Point Designer™.
See Also
Probe | Weighted Sample Time

Topics
“What Is Sample Time?”
“Specify Sample Time”
“View Sample Time Information”

Introduced before R2006a
While Iterator

Repeat execution of a subsystem while a logical expression is true

Library: Ports & Subsystems

Description

The While Iterator block, when placed in a Subsystem block, repeatedly executes the contents of the subsystem during the current time step while the value of the input condition is true or 1. Use this block to implement the block diagram equivalent of a while loop in a programming language.

The While Iterator Subsystem block is preconfigured with a While Iterator block. Placing a While Iterator block in a Subsystem block makes it an atomic subsystem.

Ports

Input

cond — Logical condition signal

scalar

Signal with the result from evaluating a logical condition. Because the subsystem is not externally triggered during a time step, evaluating a condition as true (1) or false (0) must reside within the subsystem.

The data type and values of the signal can be:

• Logical (Boolean) — true (1) or false (0).
• Numerical — true (any positive or negative number) or false (0).
Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point

**IC — Initial logical condition**

scalar

Signal with the initial logical condition. At the beginning of each time step:

- If IC is false (0), the subsystem does not execute during the time step.
- If IC is true (value not equal to 0), the subsystem starts executing and continues to repeat execution as long as the cond signal is true.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point

**Output Arguments**

**Iteration number — Signal with the number of iterations**

scalar

Signal with the number of While Iterator Subsystem block executions during each time step.

Data Types: double | int8 | int16 | int32

**Parameters**

**Maximum number of iterations — Specify maximum number of iterations**

-1 (default) | integer

Specify maximum number of iterations allowed during a time step.

-1

Any number of iterations as long as the cond signal is true (value not equal to 0). If you specify -1 and the cond signal never becomes false (0), the simulation runs in an infinite loop. In this case, the only way to stop the simulation is to terminate MATLAB.

integer

Maximum number of iterations during a time step.
Programmatic Use

Block Parameter: MaxIters
Type: character vector
Values: '5' | '-1' | '<integer>'
Default: '5'

While loop type — Select type of block
while (default) | do-while

Select type of block.

while

The While Iterator block has two inputs, a cond (logical condition) input and an IC (initial logical condition) input. The source of the IC signal must be external to the While Iterator Subsystem block.

At the beginning of each time step:

- If the IC input is true (value not equal to 0), the blocks in the subsystem repeat execution while the cond input is true. This process continues during a time step as long as the cond input is true and the number of iterations is less than or equal to the Maximum number of iterations.
- If the IC input is false, the While Iterator block does not execute the contents of the subsystem.

do-while

The While Iterator block has one input, the cond (while condition) input.

At each time step, the blocks in the subsystem repeat execution while the cond input is true (value not equal to 0). This process continues as long as the cond input is true and the number of iterations is less than or equal to the Maximum number of iterations.

Programmatic Use

Block Parameter: WhileBlockType
Type: character vector
Values: 'while' | 'do-while'
Default: 'while'

States when starting — Select block states between time steps
held (default) | reset
Select how to handle block states between time steps.

**held**

Hold block states between time steps. Block state values persist across time steps.

**reset**

Reset block states to their initial values at the beginning of each time step and before the first iteration loop.

**Programmatic Use**

**Block Parameter:** ResetStates

- **Type:** character vector
- **Values:** 'held' | 'reset'
- **Default:** 'held'

**Show iteration number port — Control display of output port**

- **clear** | **select**

Control display of output port for signal with number of block executions. The value of the signal from this port starts at 1 and is incremented by 1 for each succeeding iteration.

- **off**
  - Remove output port.

- **on**
  - Display output port for signal with iteration number.

**Dependencies**

Selecting this parameter enables the **Output data type** parameter.

**Programmatic Use**

**Block Parameter:** ShowIterationPort

- **Type:** character vector
- **Values:** 'off' | 'on'
- **Default:** 'off'

**Output data type — Select output data type for current iteration number**

- **int32 (default)** | **uint32** | **int16** | **uint16** | **int8** | **uint8** | **double**
Select output data type for iteration number signal. The value of this signal is the number of iterations during a time step and the total number of iterations at the end of a time step.

**int32**
Signed 32-bit integer.

**uint32**
Unsigned 32-bit integer.

**int16**
Signed 16-bit integer.

**uint16**
Unsigned 16-bit integer.

**int8**
Signed 8-bit integer.

**uint8**
Unsigned 8-bit integer.

**double**
Double-precision floating point.

**Dependencies**

Select the **Show iteration number port** check box to enable this parameter.

**Programmatic Use**

**Block Parameter:** OutputDataType  
**Type:** character vector  
**Value:** 'int32' | 'uint32' | 'int16' | 'uint16' | 'int8' | 'uint8' | 'double'  
**Default:** 'int32'

**See Also**

**Blocks**
Subsystem | While Iterator Subsystem

**Topics**
Iterator Subsystem Execution
Introduced before R2006a
**While Iterator Subsystem**

Subsystem that repeats execution during a simulation time step  
**Library:** Simulink / Ports & Subsystems

**Description**

The While Iterator Subsystem block is a Subsystem block preconfigured as a starting point for creating a subsystem that repeats execution during a simulation time step while a logical condition is true.

Use While Iterator Subsystem blocks to model:

- Block diagram equivalent of a program while or do-while loop.
- An iterative algorithm that converges on a more accurate solution after multiple iterations.

When using simplified initialization mode, if you place a block that needs elapsed time (such as a Discrete-Time Integrator block) in a While Iterator Subsystem block, Simulink displays an error.

If the output signal from a While Iterator Subsystem block is a function-call signal, Simulink displays an error when you simulate the model or update the diagram.
**Ports**

**Input**

**In1 — Signal input to a subsystem block**
scalar | vector | matrix

Placing an Inport block in a subsystem block adds an external input port to the block. The port label matches the name of the Inport block.

Use Inport blocks to get signals from the local environment.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus

**IC (initial logical condition) — Control initial execution of a subsystem block**
scalar

Placing a While Iterator block connected to an Input block in a Subsystem block adds this external input port to the block.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point

**Output**

**Out — Signal output from a subsystem**
scalar | vector | matrix

Placing an Outport block in a subsystem block adds an output port from the block. The port label on the subsystem block is the name of the Outport block.

Use Outport blocks to send signals to the local environment.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point | enumerated | bus
Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>bus</th>
<th>double</th>
<th>enumerated</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
<th>string</th>
</tr>
</thead>
</table>

| Direct Feedthrough  | no      |     |        |            |             |         |        |        |
| Multidimensional Signals | limited |     |        |            |             |         |        |        |
| Variable-Size Signals | limited |     |        |            |             |         |        |        |
| Zero-Crossing Detection | no      |     |        |            |             |         |        |        |

a. Actual data type or capability support depends on block implementation.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

Actual data type or capability support depends on block implementation.

Fixed-Point Conversion
Design and simulate fixed-point systems using Fixed-Point Designer™.

Actual data type support depends on block implementation.

See Also

Blocks
Subsystem | While Iterator
Topics
Iterator Subsystem Execution

Introduced before R2006a
**Width**

Output width of input vector  
**Library:** Simulink / Signal Attributes

**Description**

The Width block generates as output the width of its input vector.

You can use an array of buses as an input signal to a Width block. For details about defining and using an array of buses, see “Combine Buses into an Array of Buses”.

**Ports**

**Input**

*Port_1 — Input signal*  
scalar | vector | matrix | N-D array

Input signal specified as a scalar, vector, matrix, or N-D array.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `Boolean` | `fixed point` | `enumerated` | `bus`

**Output**

*Port_1 — Width of input signal*  
scalar

Output is the width of the input signal, specified as a scalar.
Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fixed point

Parameters

Output data type mode — Output data type mode
Choose intrinsic data type (default) | Inherit via back propagation | All ports same datatype

Specify the output data type to be the same as the input, or inherit the data type by back propagation. You can also choose to specify a built-in data type from the drop-down list in the Output data type parameter.

Programmatic Use
Block Parameter: OutputDataTypeScalingMode
Type: character vector
Values: 'Choose intrinsic data type' | 'Inherit via back propagation' | 'All ports same datatype'
Default: 'Choose intrinsic data type'

Output data type — Output data type
double (default) | single | int8 | uint8 | int16 | uint16 | int32 | uint32

This parameter is visible when you select Choose intrinsic data type for Output data type mode. Select a built-in data type from the drop-down list.

Programmatic Use
Block Parameter: DataType
Type: character vector
Values: 'double' | 'single' | 'int8' | 'uint8' | 'int16' | 'uint16' | 'int32' | 'uint32'
Default: 'double'

Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>bus</th>
<th>double</th>
<th>enumerated</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
<th>string</th>
</tr>
</thead>
</table>
### Extended Capabilities

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>yes</td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>yes</td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>yes</td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
</tr>
</tbody>
</table>

**C/C++ Code Generation**
Generate C and C++ code using Simulink® Coder™.

**Fixed-Point Conversion**
Design and simulate fixed-point systems using Fixed-Point Designer™.

**See Also**
Probes

**Topics**
“Variable-Size Signal Basics”

**Introduced before R2006a**
Wrap To Zero

Set output to zero if input is above threshold

Library:
Simulink / Discontinuities
HDL Coder / Discontinuities

Description

The Wrap To Zero block sets the output to zero when the input is above the **Threshold** value. When the input is less than or equal to the **Threshold**, then the output is equal to the input.

Ports

Input

Port_1 — Input signal
scalar | vector

Input signal, specified as a scalar or vector. Signal values equal to or greater than the value of **Threshold** are set to zero.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point

Output

Port_1 — Output signal
scalar | vector

Output signal set to the input signal value or zero. The data type of the output is the same data type as the input.
Tip  If the input data type cannot represent zero, parameter overflow occurs. To detect this overflow, go to the Diagnostics > Data Validity pane of the Configuration Parameters dialog box and set Parameters > Detect overflow to warning or error.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point

Parameters

To edit the parameters for the Wrap to Zero block, double-click the block icon.

Threshold — Threshold for outputting zero
255 (default) | scalar

Threshold value for setting the output value to zero.

Programmatic Use
Block Parameter: Threshold
Type: character vector
Values: scalar
Default: '255'

Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>double</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1-2351
Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture
This block has a single, default HDL architecture.

HDL Block Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConstrainedOutputPipeline</td>
<td>Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. For more details, see “ConstrainedOutputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td>InputPipeline</td>
<td>Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “InputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td>OutputPipeline</td>
<td>Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “OutputPipeline” (HDL Coder).</td>
</tr>
</tbody>
</table>

Restrictions

The input signal and Threshold parameter must have equal size. For example, if the input is a two-dimensional vector, Threshold must also be a two-dimensional vector.
**PLC Code Generation**
Generate Structured Text code using Simulink® PLC Coder™.

**Fixed-Point Conversion**
Design and simulate fixed-point systems using Fixed-Point Designer™.

**See Also**
Saturation | Saturation Dynamic

*Introduced before R2006a*
XY Graph

Display X-Y plot of signals using MATLAB figure window

Library:
- Simulink / Sinks
- HDL Coder / Sinks

Description

The XY Graph block displays an X-Y plot of its inputs in a MATLAB figure window.

The block has two scalar inputs. The block plots data from the first input (the x direction) against data from the second input (the y direction). (See “Port Location After Rotating or Flipping” for a description of the port order for various block orientations.) This block is
useful for examining limit cycles and other two-state data. Data outside the specified range does not appear.

A figure window appears for each XY Graph block in the model at the start of simulation.

**Note** The XY Graph block does not support stepping back in a simulation.

### Ports

**Input**

**Port_1 — X-axis values**

scalar

Plot input as x values on an X-Y plot. See “Port Location After Rotating or Flipping” for a description of the port order for various block orientations.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point

**Port_2 — Y-axis values**

scalar

Plot input as y values on an X-Y plot. See “Port Location After Rotating or Flipping” for a description of the port order for various block orientations.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean | fixed point

### Parameters

**X-min — Minimum x**

-1 (default) | real number

Specify the minimum x-axis value. Data below the minimum x is ignored.

**Programmatic Use**

**Block Parameter:** xmin
**Type:** character vector  
**Values:** real number  
**Default:** '-1'

**X-max — Maximum x**  
1 (default) | real number  

Specify the maximum x-axis value. Data above the maximum x is ignored.

**Programmatic Use**  
**Block Parameter:** xmax  
**Type:** character vector  
**Values:** real number  
**Default:** '1'

**Y-min — Minimum y**  
-1 (default) | real number  

Specify the minimum y-axis value. Data below the minimum y is ignored.

**Programmatic Use**  
**Block Parameter:** ymin  
**Type:** character vector  
**Values:** real number  
**Default:** '-1'

**Y-max — Maximum y**  
1 (default) | real number  

Specify the maximum y-axis value. Data above the maximum y is ignored.

**Programmatic Use**  
**Block Parameter:** ymax  
**Type:** character vector  
**Values:** real number  
**Default:** '1'

**Sample time — Sample time**  
-1 (default) | positive number  

Specify the time interval between samples. To inherit the sample time, set this parameter to -1. For more information, see “Specify Sample Time”.

1-2356
Programmatic Use
Block Parameter: st
Type: character vector
Values: '-1' (for inherited) | positive number
Default: '-1'

Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>double</th>
<th>enumerated</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

This block can be used for simulation visibility in systems that generate code, but is not included in the generated code.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

This block can be used for simulation visibility in subsystems that generate HDL code, but is not included in the hardware implementation.
**PLC Code Generation**
Generate Structured Text code using Simulink® PLC Coder™.

This block can be used for simulation visibility in systems that generate PLC code, but is not included in the generated code.

**Fixed-Point Conversion**
Design and simulate fixed-point systems using Fixed-Point Designer™.

This block accepts fixed-point input, but converts it to `double` for display.

**See Also**
Scope

**Topics**
“Decide How to Visualize Simulation Data”
“Get Started with ROS in Simulink®” (ROS Toolbox)

**Introduced before R2006a**
Zero-Order Hold

Implement zero-order hold sample period
Library: Simulink / Discrete
HDL Coder / Discrete

Description

The Zero-Order Hold block holds its input for the sample period you specify. If the input is a vector, the block holds all elements of the vector for the same sample period.

You specify the time between samples with the Sample time parameter. A setting of -1 means the block inherits the Sample time.

**Tip** Do not use the Zero-Order Hold block to create a fast-to-slow transition between blocks operating at different sample rates. Instead, use the Rate Transition block.

Bus Support

The Zero-Order Hold block is a bus-capable block. The input can be a virtual or nonvirtual bus signal. No block-specific restrictions exist. All signals in a nonvirtual bus input to a Zero-Order Hold block must have the same sample time, even if the elements of the associated bus object specify inherited sample times. You can use a Rate Transition block to change the sample time of an individual signal, or of all signals in a bus. See “Specify Bus Signal Sample Times” and “Bus-Capable Blocks” for more information.

You can use an array of buses as an input signal to a Zero-Order Hold block. For details about defining and using an array of buses, see “Combine Buses into an Array of Buses”.

Comparison with Similar Blocks

The Memory, Unit Delay, and Zero-Order Hold blocks provide similar functionality but have different capabilities. Also, the purpose of each block is different.
This table shows recommended usage for each block.

<table>
<thead>
<tr>
<th>Block</th>
<th>Purpose of the Block</th>
<th>Reference Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unit Delay</td>
<td>Implement a delay using a discrete sample time that you specify. The block accepts and outputs signals with a discrete sample time.</td>
<td>• sldemo_enginewc (Compression subsystem)</td>
</tr>
<tr>
<td>Memory on page 1-1278</td>
<td>Implement a delay by one major integration time step. Ideally, the block accepts continuous (or fixed in minor time step) signals and outputs a signal that is fixed in minor time step.</td>
<td>• sldemo_bounce</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• sldemo_clutch (Friction Mode Logic/Lockup FSM subsystem)</td>
</tr>
<tr>
<td>Zero-Order Hold</td>
<td>Convert an input signal with a continuous sample time to an output signal with a discrete sample time.</td>
<td>• sldemo_radar_eml</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• aero_dap3dof</td>
</tr>
</tbody>
</table>

Each block has the following capabilities.

<table>
<thead>
<tr>
<th>Capability</th>
<th>Memory</th>
<th>Unit Delay</th>
<th>Zero-Order Hold</th>
</tr>
</thead>
<tbody>
<tr>
<td>Specification of initial condition</td>
<td>Yes</td>
<td>Yes</td>
<td>No, because the block output at time t = 0 must match the input value.</td>
</tr>
<tr>
<td>Specification of sample time</td>
<td>No, because the block can only inherit sample time from the driving block or the solver used for the entire model.</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Support for frame-based signals</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Support for state logging</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>
Ports

Input

Port_1 — Input signal
scalar | vector

Input signal that the block holds by one sample period.

Data Types:
single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 |
Boolean | fixed point | enumerated | bus

Output

Port_1 — Output signal
scalar | vector

Output signal that is the input held by one sample period.

Data Types:
single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 |
Boolean | fixed point | enumerated | bus

Parameters

Sample time (-1 for inherited) — Discrete interval between sample time hits
-1 (default) | scalar

Specify the time interval between samples. To inherit the sample time, set this parameter to -1. See “Specify Sample Time” in the online documentation for more information.

Do not specify a continuous sample time, either 0 or [0, 0]. This block supports only discrete sample times. When this parameter is -1, the inherited sample time must be discrete and not continuous.
Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Boolean</th>
<th>bus</th>
<th>double</th>
<th>enumerated</th>
<th>fixed point</th>
<th>integer</th>
<th>single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

Generated code relies on memcpy or memset functions (strong.h) under certain conditions.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture
This block has a single, default HDL architecture.
HDL Block Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConstrainedOutputPipeline</td>
<td>Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. For more details, see “ConstrainedOutputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td>InputPipeline</td>
<td>Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “InputPipeline” (HDL Coder).</td>
</tr>
<tr>
<td>OutputPipeline</td>
<td>Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “OutputPipeline” (HDL Coder).</td>
</tr>
</tbody>
</table>

Complex Data Support

This block supports code generation for complex signals.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

Fixed-Point Conversion

Design and simulate fixed-point systems using Fixed-Point Designer™.

See Also

Memory | Unit Delay

Introduced before R2006a
Zero-Pole

Model system by zero-pole-gain transfer function

Library: Simulink / Continuous

Description

The Zero-Pole block models a system that you define with the zeros, poles, and gain of a Laplace-domain transfer function. This block can model single-input single-output (SISO) and single-input multiple-output (SIMO) systems.

Conditions for Using This Block

The Zero-Pole block assumes the following conditions:

• The transfer function has the form

\[ H(s) = \frac{KZ(s)}{P(s)} = \frac{(s-Z(1))(s-Z(2)) \ldots (s-Z(m))}{(s-P(1))(s-P(2)) \ldots (s-P(n))} , \]

where \( Z \) represents the zeros, \( P \) the poles, and \( K \) the gain of the transfer function.

• The number of poles must be greater than or equal to the number of zeros.

• If the poles and zeros are complex, they must be complex-conjugate pairs.

• For a multiple-output system, all transfer functions must have the same poles. The zeros can differ in value, but the number of zeros for each transfer function must be the same.

Note You cannot use a Zero-Pole block to model a multiple-output system when the transfer functions have a differing number of zeros or a single zero each. Use multiple Zero-Pole blocks to model such systems.
Modeling a Single-Output System

For a single-output system, the input and the output of the block are scalar time-domain signals. To model this system:

1. Enter a vector for the zeros of the transfer function in the **Zeros** field.
2. Enter a vector for the poles of the transfer function in the **Poles** field.
3. Enter a 1-by-1 vector for the gain of the transfer function in the **Gain** field.

Modeling a Multiple-Output System

For a multiple-output system, the block input is a scalar and the output is a vector, where each element is an output of the system. To model this system:

1. Enter a matrix of zeros in the **Zeros** field.
   
   Each *column* of this matrix contains the zeros of a transfer function that relates the system input to one of the outputs.

2. Enter a vector for the poles common to all transfer functions of the system in the **Poles** field.

3. Enter a vector of gains in the **Gain** field.
   
   Each element is the gain of the corresponding transfer function in **Zeros**.

   Each element of the output vector corresponds to a column in **Zeros**.

Transfer Function Display on the Block

The Zero-Pole block displays the transfer function depending on how you specify the zero, pole, and gain parameters.

- If you specify each parameter as an expression or a vector, the block shows the transfer function with the specified zeros, poles, and gain. If you specify a variable in parentheses, the block evaluates the variable.

For example, if you specify **Zeros** as \([3,2,1]\), **Poles** as \((\text{poles})\), where \(\text{poles}\) is \([7,5,3,1]\), and **Gain** as \(\text{gain}\), the block looks like this.

\[
\frac{\text{gain}(s^3+s^2+s)}{(s^7+s^5+s^3+s)}
\]
• If you specify each parameter as a variable, the block shows the variable name followed by (s) if appropriate.

For example, if you specify Zeros as zeros, Poles as poles, and Gain as gain, the block looks like this.

\[
\frac{\text{gain} \cdot \text{zeros}(s)}{\text{poles}(s)}
\]

### Ports

#### Input

**Port_1 — Input signal**

scalar

Input signal, specified as a scalar with data type double.

Data Types: double

#### Output

**Port_1 — Output signal**

scalar | vector

System modeled by a zero-pole gain transfer function, provided as a scalar or vector signal with data type double.

- When modeling a single-output system, the block outputs a scalar time-domain signal. For more information, see “Modeling a Single-Output System” on page 1-2365.
- When modeling a multiple-output system, the block outputs a vector, where each element is an output of the system. For more information, see “Modeling a Multiple-Output System” on page 1-2365.

Data Types: double
Parameters

**Zeros — Matrix of zeros**

[1] (default) | vector | matrix

Define the matrix of zeros.

- For a single-output system, enter a vector for the zeros of the transfer function.
- For a multiple-output system, enter a matrix. Each column of this matrix contains the zeros of a transfer function that relates the system input to one of the outputs.

**Programmatic Use**

**Block Parameter:** Zeros

**Type:** character vector, string

**Value:** vector | matrix

**Default:** ' [1] '

**Poles — Vector of poles**

[0 -1] (default) | vector

Define the vector of poles.

- For a single-output system, enter a vector for the poles of the transfer function.
- For a multiple-output system, enter a vector for the poles common to all transfer functions of the system.

**Programmatic Use**

**Block Parameter:** Poles

**Type:** character vector, string

**Value:** vector

**Default:** ' [0 -1] '

**Gain — Vector of gains**

[1] (default) | vector

Define the vector of gains.

- For a single-output system, enter a 1-by-1 vector for the gain of the transfer function.
- For a multiple-output system, enter a vector of gains. Each element is the gain of the corresponding transfer function in **Zeros**.
Programmatic Use
Block Parameter: Gain
Type: character vector, string
Value: vector
Default: '[1]' 

Absolute tolerance — Absolute tolerance for computing block states
auto (default) | scalar | vector

Absolute tolerance for computing block states, specified as a positive, real-valued, scalar or vector. To inherit the absolute tolerance from the Configuration Parameters, specify auto or -1.

- If you enter a real scalar, then that value overrides the absolute tolerance in the Configuration Parameters dialog box for computing all block states.
- If you enter a real vector, then the dimension of that vector must match the dimension of the continuous states in the block. These values override the absolute tolerance in the Configuration Parameters dialog box.
- If you enter auto or -1, then Simulink uses the absolute tolerance value in the Configuration Parameters dialog box (see “Solver Pane”) to compute block states.

Programmatic Use
Block Parameter: AbsoluteTolerance
Type: character vector, string
Values: 'auto' | '-1' | any positive real-valued scalar or vector
Default: 'auto'

State Name (e.g., 'position') — Assign unique name to each state
' ' (default) | 'position' | {'a', 'b', 'c'} | a | ...

Assign a unique name to each state. If this field is blank (' '), no name assignment occurs.

- To assign a name to a single state, enter the name between quotes, for example, 'position'.
- To assign names to multiple states, enter a comma-delimited list surrounded by braces, for example, {'a', 'b', 'c'}. Each name must be unique.
- To assign state names with a variable in the MATLAB workspace, enter the variable without quotes. A variable can be a character vector, string, cell array, or structure.
Limitations

- The state names apply only to the selected block.
- The number of states must divide evenly among the number of state names.
- You can specify fewer names than states, but you cannot specify more names than states.

For example, you can specify two names in a system with four states. The first name applies to the first two states and the second name to the last two states.

Programmatic Use

**Block Parameter:** ContinuousStateAttributes

**Type:** character vector, string

**Values:** ' ' | user-defined

**Default:** ' '

Block Characteristics

<table>
<thead>
<tr>
<th>Data Types</th>
<th>double</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Feedthrough</td>
<td>yes</td>
</tr>
<tr>
<td>Multidimensional Signals</td>
<td>no</td>
</tr>
<tr>
<td>Variable-Size Signals</td>
<td>no</td>
</tr>
<tr>
<td>Zero-Crossing Detection</td>
<td>no</td>
</tr>
</tbody>
</table>

Extended Capabilities

**C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.
Not recommended for production-quality code. Relates to resource limits and restrictions on speed and memory often found in embedded systems. The code generated can contain dynamic allocation and freeing of memory, recursion, additional memory overhead, and widely-varying execution times. While the code is functionally valid and generally acceptable in resource-rich environments, smaller embedded targets often cannot support such code.

In general, consider using the Simulink Model Discretizer to map continuous blocks into discrete equivalents that support production code generation. To start the Model Discretizer, select Analysis > Control Design > Model Discretizer. One exception is the Second-Order Integrator block because, for this block, the Model Discretizer produces an approximate discretization.

See Also
Discrete Zero-Pole

Topics
“State”

Introduced before R2006a
Functions — Alphabetical List
**add_block**

Add block to model

**Syntax**

```matlab
h = add_block(source,dest)
h = add_block(source,dest,'MakeNameUnique','on')
h = add_block(___,'CopyOption','nolink')
h = add_block(sourceIn,destIn,'CopyOption','duplicate')
h = (___,Name,Value)
```

**Description**

`h = add_block(source,dest)` adds a copy of the block `source` from a library or model to the specified destination model and block name. This syntax creates the block at the same location as it appears in the model or the library model.

If you are copying between models or from a library, load the destination model first.

`h = add_block(source,dest,'MakeNameUnique','on')` ensures that the destination block name is unique in the model. This syntax adds a number to the destination block name if a block with that name exists, incrementing to ensure a unique name.

`h = add_block(___,'CopyOption','nolink')` copies the block or subsystem `source` from a library without creating a link to the library block.

`h = add_block(sourceIn,destIn,'CopyOption','duplicate')` duplicates an inport block in a subsystem, giving the destination block the same port number as the source block. Duplicate an inport to branch a signal from an input port without creating a port or adding lines. For more information, see “Creating Duplicate Inports” on page 1-1000.

`h = (___,Name,Value)` uses optional `Name,Value` arguments.

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes.
You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

With the add_block function, you can use block parameter and value pairs. For a list of all the block parameters, see “Common Block Properties” on page 6-112 and “Block-Specific Parameters” on page 6-132.

**Examples**

**Add Block to Model from a Library**

Add the block from the Simulink library to the model f14.

Load or open the destination model.

```matlab
open_system('f14');
```

Add the Scope block from the Simulink Sinks library to f14, naming the new block MyScope.

```matlab
add_block('simulink/Sinks/Scope','f14/MyScope');
```

**Add a Block from Another Model**

Add a copy of a block from the model f14 to vdp.

Load or open the destination model.

```matlab
open_system('vdp');
```

Add the Actuator Model block from f14 to vdp.

```matlab
add_block('f14/Actuator Model','vdp/Actuator Model');
```
### Add a Block Using a Unique Name

Add the block from the Simulink library to the model `vdp`. Because there is already a block named Scope in `vdp`, use the `MakeNameUnique` option to ensure that the new block name is unique.

Load or open the destination model.

```matlab
open_system('vdp');
```

Add the Scope block from the Simulink Sinks library to `vdp`, ensuring that the name is unique.

```matlab
add_block('simulink/Sinks/Scope','vdp/Scope','MakeNameUnique','on')
```

### Duplicate an Inport Block in a Subsystem

Add an inport block in the `f14/Controller` subsystem that uses the same port number as another inport in that subsystem.

Duplicate the Stick Input (in) block in the Controller subsystem, naming the copy Stick Input (in)2. The resulting block uses the same port number as Stick Input (in) but does not add an inport on the parent subsystem. The signal that enters that port branches to both inports.

```matlab
add_block('f14/Controller/Stick Input (in)',...
 'f14/Controller/Stick Input (in)2','CopyOption','duplicate')
```

### Add a Block and Set Parameters

Add a block from a library to a model and set parameters using a Name,Value pair.

Load or open the destination model.

```matlab
open_system('vdp');
```

Add a Gain block from the library to `vdp`, and set the Gain value to 5.
add_block('simulink/Math Operations/Gain','vdp/Five','Gain','5')

**Input Arguments**

**source — Block to copy to model**

block path name | library block path name

Block to copy to model, specified as:

- The full block path name if you are copying the block from another model, for example, 'vdp/Mu'. This usage copies the block and its settings.
- The library block path name if you want to add a block from a library, for example, 'simulink/Math Operations/Gain'.

To get the library block path name, you can hover over the block in the Library Browser. Alternatively, you can open the library model, select the block, and enter `gcb` at the command line. To open the library model, in the Library Browser, right-click the library name in the library list and select *Open library_name library*.

You can also use the syntax `built-in/blocktype` as the source block path name, where `blocktype` is the programmatic block name—the value of the BlockType parameter (see “Common Block Properties” on page 6-112). However, blocks added using `built-in/blocktype` sometimes have different default parameter values from library blocks.

For subsystems and masked blocks, use the library block path name. Using the BlockType value (SubSystem) creates an empty subsystem.

Example: 'vdp/Mu','simulink/Sinks/Scope'

**dest — Name and location of new block**

block path name

Name and location of the new block in the model, specified as the block path name.

Example: 'f14/Controller/MyNewBlock'

**sourceIn — Inport block whose port number to copy**

block path name

Inport block whose port number to copy, specified as the block path name.
Example: 'f14/Controller/Stick Input (in)', 'myModel/mySubsystem/In1'

destIn — Inport block to create
block path name

Inport block with duplicate port number to create, specified as the block path name. Create the destination block in the same system as the source block.

Example: 'myModel/mySubsystem/DupPortIn'

Output Arguments

h — New block
handle

New block, returned as a handle.

See Also

delete_block

Topics

“Delete an Annotation Programmatically”
“Create Annotations Programmatically”

Introduced before R2006a
add_exec_event_listener

Register listener for block method execution event

Syntax

\[
h = \text{add_exec_event_listener}(blk, \text{event}, \text{listener});\]

Description

\[
h = \text{add_exec_event_listener}(blk, \text{event}, \text{listener})\]

registers a listener for a block method execution event where the listener is a MATLAB program that performs some task, such as logging runtime data for a block, when the event occurs (see “Listen for Method Execution Events”). Simulink software invokes the registered listener whenever the specified event occurs during simulation of the model. You cannot register a listener for virtual blocks.

**Note** Simulink software can register a listener only while a simulation is running. Invoking this function when no simulation is running results in an error message. To ensure that a listener catches all relevant events triggered by a model’s simulation, you should register the listener in the model's StartFcn callback function (see “Callbacks for Customized Model Behavior”).

Input Arguments

**blk**

Specifies the block whose method execution event the listener is intended to handle. May be one of the following:

- Full pathname of a block
- A block handle
- A block runtime object (see “Access Block Data During Simulation”.)
event
Specifies the type of event for which the listener listens. It may be any of the following:

<table>
<thead>
<tr>
<th>Event</th>
<th>Occurs...</th>
</tr>
</thead>
<tbody>
<tr>
<td>'PreDerivatives'</td>
<td>Before a block's Derivatives method executes</td>
</tr>
<tr>
<td>'PostDerivatives'</td>
<td>After a block's Derivatives method executes</td>
</tr>
<tr>
<td>'PreOutputs'</td>
<td>Before a block's Outputs method executes</td>
</tr>
<tr>
<td>'PostOutputs'</td>
<td>After a block's Outputs method executes</td>
</tr>
<tr>
<td>'PreUpdate'</td>
<td>Before a block's Update method executes</td>
</tr>
<tr>
<td>'PostUpdate'</td>
<td>After a block's Update method executes</td>
</tr>
</tbody>
</table>

listener
Specifies the listener to be registered. It may be either a character vector specifying a MATLAB expression, e.g., 'disp(''here'')' or a handle to a MATLAB function that accepts two arguments. The first argument is the block runtime object of the block that triggered the event. The second argument is an instance of EventData class that specifies the runtime object and the name of the event that just occurred.

Output Arguments

add_exec_event_listener returns a handle to the listener that it registered. To stop listening for an event, use the MATLAB clear command to clear the listener handle from the workspace in which the listener was registered.

Introduced before R2006a
add_line

Add line to Simulink model

Syntax

h = add_line(sys,out,in)
h = add_line(sys,out,in,'autorouting',autoOption)
h = add_line(sys,points)

Description

h = add_line(sys,out,in) adds a line in the model or subsystem sys that connects one block's output port out to another block's input port in. This syntax draws the most direct route from port to port, for example, diagonal lines or lines that go through other blocks.

You can connect ports when:

• The input port does not already have a connection.
• The blocks are compatible for connecting.

h = add_line(sys,out,in,'autorouting',autoOption) connects blocks, specifying whether to route the lines around other blocks.

h = add_line(sys,points) adds a line drawn by (x,y) coordinate points relative to the upper-left corner of the Simulink Editor canvas before any canvas resizing. If either end of the line is within five pixels of a corresponding port, the function connects the line to it. The line can have multiple segments.

Examples

Connect Blocks Using Port Numbers

Use the block port numbers to add a line to connect blocks.
Create a model and open it.

```matlab
open_system(new_system('connect_model'));
```

Add and position a Constant block and a Gain block.

```matlab
add_block('simulink/Commonly Used Blocks/Constant','connect_model/Constant');
set_param('connect_model/Constant','position',[140,80,180,120]);
add_block('simulink/Commonly Used Blocks/Gain','connect_model/Gain');
set_param('connect_model/Gain','position',[220,80,260,120]);
```

Connect the blocks. Each block has one port, so specify port 1.

```matlab
add_line('connect_model','Constant/1','Gain/1');
```

**Connect Blocks Using Port Handles**

Get the port handles and connect the ports using `add_line`.

Open the model `vdp`.

```matlab
open_system('vdp');
```

Delete the line that connects the Mu gain block to the Sum block.

```matlab
delete_line('vdp','Mu/1','Sum/2');
```

Get the port handles from the Mu block and the Sum block.

```matlab
h = get_param('vdp/Mu','PortHandles');
h1 = get_param('vdp/Sum','PortHandles');
```

Look at the `h1` structure. Notice the two handles for the Inport property.

```matlab
h1
```

```matlab
h1 =

struct with fields:

    Inport: [47.0002 54.0002]
    Outport: 39.0002
    Enable: []
    Trigger: []
    State: []
Index into the Outport and Inport properties on the port handles to get the handles you want and connect them. Connect to the second inport.

```matlab
add_line('vdp',h.Outport(1),h1.Inport(2));
```

**Add a Branched Line**

You can branch a line by adding a connection programmatically. You can use the `points` syntax to draw the segment, or you can draw the line by specifying the ports to connect. When using the port, use automatic line routing to improve the look of the branched line.

Add a scope to the `vdp` model above the outport.

```matlab
vdp
add_block('simulink/Commonly Used Blocks/Scope','vdp/Scope1');
set_param('vdp/Scope1','position',[470,70,500,110]);
```

Connect the Integrator block `x1` to `Scope1`. This code branches the existing line from the `x1` output and connects it to the scope. With autorouting on, the resulting line is segmented.

```matlab
add_line('vdp','x1/1','Scope1/1','autorouting','on');
```

**Connect Blocks Using Points**

You can use points on the canvas as the start and end of each segment. Get the port locations using `get_param` with the 'PortConnectivity' option.

Open the model `vdp` and delete the line that connects the Mu and Sum blocks.

```matlab
vdp
delete_line('vdp','Mu/1','Sum/2')
```

Get the port locations for Mu. Mu has two ports. The first is the input port, and the second is the output port.
mu = get_param('vdp/Mu','PortConnectivity');
mu.Position
ans =
  190   150

ans =
  225   150

Get the port locations for Sum, which has three ports. The second position is the lower input port.
s = get_param('vdp/Sum','PortConnectivity');
s.Position
ans =
  250   135

ans =
  250   150

ans =
  285   145

Connect the ports using the output and input points.
add_line('vdp',[225 150; 250 150])

**Connect Blocks Using Autorouting Options**

This example shows the effect of adding lines with and without autorouting options.

Create a model route. Display default block names.

open_system(new_system('route'));
set_param('route','HideAutomaticNames','off')
Add blocks as shown. Add an inport and outport to each subsystem.

Add lines to connect the outputs from Subsystem to the inputs of Subsystem1.

```matlab
add_line('route', {'Subsystem/1', 'Subsystem/2'}, ...
            {'Subsystem1/1', 'Subsystem1/2'})
```

Because you did not use the autorouting options, the function draws straight lines, which pass through the Gain block.

Delete the lines. Add lines again, this time using the autorouting option set to 'on'.

```matlab
add_line('route', {'Subsystem/1', 'Subsystem/2'}, ...
            {'Subsystem1/1', 'Subsystem1/2'}, 'autorouting', 'on')
```

The lines route around the Gain block.
Delete the lines. Add lines again, using the `smart` autorouting option. When you use an array to connect two sets of inports and outports, 'smart' autorouting routes them together if doing so makes better use of the space.

```matlab
add_line('route',{Subsystem/1,Subsystem/2},...
    {Subsystem1/1,Subsystem1/2},'autorouting','smart')
```

### Input Arguments

**sys** — Model or subsystem to add line to  
character vector

Model or subsystem to add the line to, specified as character vector.

Example: `'vdp'`, `'f14/Controller'`

**out** — Block output port to connect line from  
block/port name or number character vector | port handle | array of port designators

Block output port to connect line from, specified as:

- The block name, a slash, and the port name or number. Most block ports are numbered from top to bottom or from left to right. For a state port, use the port name `State` instead of a port number.
- The port handle that you want to connect from.
- An array of either of these port designators.

Use `'PortHandles'` with `get_param` to get the handles.

Example: `'Mu/1'`, `'Subsystem/2', h.Outport(1){"Subsystem/1', 'Subsystem/2'}`
**in — Block input port to connect line to**

Block input port to connect line to, specified as:

- The block name, a slash, and the port name or number. The port name on:
  - An enabled subsystem is Enable.
  - A triggered subsystem is Trigger.
  - If Action and Switch Case Action subsystems is Action.
- The port handle that you want to add the line to.
- An array of either of these port designators.

Use the 'PortHandles' option with `get_param` to get handles.

Example: `Mu/1`, `Subsystem/2`, `h.Inport(1), {'Subsystem/1','Subsystem/2'}`

**autoOption — Type of automatic line routing**

'off' (default) | 'on' | 'smart'

Type of automatic line routing around other blocks, specified as:

- 'off' for no automatic line routing
- 'on' for automatic line routing
- 'smart' for automatic line routing that takes the best advantage of the blank spaces on the canvas and avoids overlapping other lines and labels

**points — Points of the line to draw**

matrix

Points of the line to draw, specified as at least a 2-by-2 matrix. Add a row for every segment you want to draw. Specify points as (x,y) coordinates from the upper-left corner of the Editor before any canvas resizing.

Example: `[100 300; 200 300; 200 300; 200 500]`
Output Arguments

h — Line handle

Line created by add_line, returned as a handle.

See Also
add_block | delete_block | delete_line | get_param | set_param

Topics
“Create an Enabled Subsystem”
“Create a Triggered Subsystem”

Introduced before R2006a
add_param

Add parameter to Simulink system

Syntax

add_param('sys','parameter1',value1,'parameter2',value2,...)

Description

The add_param command adds the specified parameters to the specified system and initializes the parameters to the specified values. Case is ignored for parameter names. Value character vectors are case sensitive. The value of the parameter must be a character vector. Once the parameter is added to a system, set_param and get_param can be used on the new parameters as if they were standard Simulink parameters. Simulink software saves these new parameters with the model file.

Note If you attempt to add a parameter that has the same name as an existing parameter of the system, Simulink software displays an error.

Examples

This command

add_param('vdp','DemoName','VanDerPolEquation','EquationOrder','2')

adds the parameters DemoName and EquationOrder with 'VanDerPolEquation' and '2' to the vdp system. Afterward, you can use the following command to retrieve the value of the DemoName parameter.

get_param('vdp','DemoName')

See Also

delete_param|get_param|set_param
Introduced before R2006a
addFile

(Not recommended) Add file to project

**Note**  *simulinkproject* is not recommended. Use *currentProject* or *openProject* instead. For more information, see “Compatibility Considerations”.

**Syntax**

```matlab
addFile(proj, fileorfolder)
```

**Description**

`addFile(proj, fileorfolder)` adds a file to the project `proj`.

**Examples**

**Add Files to a Project**

Open the airframe project and create a project object.

```matlab
sldemo_slproject_airframe
proj = simulinkproject;
```

Remove a file.

```matlab
removeFile(proj,'models/AnalogControl.slx')
```

Add the file back to the project.

```matlab
addFile(proj,'models/AnalogControl.slx');
```

Create and save a new model.

```matlab
new_system('mymodel');
save_system('mymodel');
```
Add the new file to the project and return a project file object.
newPrjFile = addFile(proj,'mymodel.slx');

Use the project file object to manipulate the file, for example, adding a label.
addLabel(newPrjFile, 'Classification', 'Design')

Input Arguments

proj — Project
description as a project object. Use simulinkproject to create a project object to manipulate a project at the command line.

fileorfolder — Path of file or folder
description as a character vector | cell array of character vectors | string array

Path of the file or folder to add relative to the project root folder, specified as a character vector, string, or array. Files must include the file extension. The file or folder must be within the root folder.

Example: ‘models/myModelName.slx’

Compatibility Considerations

Simulink project API is not recommended
Not recommended starting in R2019a

Starting in R2019a, instead of simulinkproject and related functions, use the currentProject or openProject functions in MATLAB. The Simulink project API will continue to be supported, but, after R2019a, new features might be available only if you use the new MATLAB project API. There are no plans to remove the Simulink project API at this time.

The new MATLAB project API is part of the R2019a functionality enabling you to use projects in MATLAB, with or without Simulink. You can now share projects with users who do not have Simulink.
See Also

Functions
addFolderIncludingChildFiles | removeFile | simulinkproject

Introduced in R2013a
addFolderIncludingChildFiles

(Not recommended) Add folder and child files to project

**Note** `simulinkproject` is not recommended. Use `currentProject` or `openProject` instead. For more information, see “Compatibility Considerations”.

**Syntax**

```matlab
addFolderIncludingChildFiles(proj, folder)
```

**Description**

`addFolderIncludingChildFiles(proj, folder)` adds a folder and all child files to the project `proj`.

**Examples**

**Add Folders to a Project**

Open the airframe project and create a project object.

```matlab
sldemo_slproject_airframe
proj = simulinkproject;
```

Create a new folder in the project folder.

```matlab
new_folder_path = fullfile(proj.RootFolder, 'new_folder')
mkdir(new_folder_path);
```

Create a new folder in the previous folder.

```matlab
new_sub_folder_path = fullfile(new_folder_path, 'new_sub_folder')
mkdir(new_sub_folder_path);
```
Create a new file in the folder.

```matlab
filepath = fullfile(new_sub_folder_path, 'new_model_in_subfolder.slx')
new_system('new_model_in_subfolder');
save_system('new_model_in_subfolder', filepath)
```

Add this new folder and child files to the project.

```matlab
projectFile = addFolderIncludingChildFiles(proj, new_folder_path)
```

**Input Arguments**

- `proj` — Project
  project object

  Project, specified as a project object. Use `simulinkproject` to create a project object to manipulate a project at the command line.

- `folder` — Path of folder
  character vector | string

  Path of the folder to add relative to the project root folder, specified as a character vector or string. The folder must be within the root folder.

  Example: ‘models’

**Compatibility Considerations**

**Simulink project API is not recommended**

*Not recommended starting in R2019a*

Starting in R2019a, instead of `simulinkproject` and related functions, use the `currentProject` or `openProject` functions in MATLAB. The Simulink project API will continue to be supported, but, after R2019a, new features might be available only if you use the new MATLAB project API. There are no plans to remove the Simulink project API at this time.

The new MATLAB project API is part of the R2019a functionality enabling you to use projects in MATLAB, with or without Simulink. You can now share projects with users who do not have Simulink.
See Also

Functions
addFile | removeFile | simulinkproject

Introduced in R2015b
addterms

Add terminators to unconnected ports in model

Syntax

addterms('sys')

Description

addterms('sys') adds Terminator and Ground blocks to the unconnected ports in the Simulink block diagram sys.

See Also

slupdate

Introduced before R2006a
attachComponent

Attach a component to a configuration set

Syntax

attachComponent(cs,component)

Description

attachComponent(cs,component) attaches a component to a Simulink.ConfigSet object.

Examples

Replace Solver Component for Active Configuration Set

Replace the solver component of the active configuration set of one model with the solver component of another model.

Get the active configuration set for modelB.

hCs = getActiveConfigSet('modelB');

Get the 'Solver' component for this configuration set.

hSolverConfig = getComponent(hCs,'Solver');

Create a copy of the component.

hSolverConfig = copy(hSolverConfig);

Get the active configuration set for modelA.

hCs = getActiveConfigSet('modelA');

Attach the copy of the 'Solver' component from modelB to modelA.
attachComponent(hCs,hSolverConfig);

**Input Arguments**

- **cs** — Configuration set object  
  *ConfigSet object*  
  A configuration set object that you can attach a component to.

- **component** — Component object  
  *SimulinkConfigComponent object*  
  A component that you can attach to configuration set.

**See Also**

- *Simulink.ConfigSet*

**Topics**

- “About Configuration Sets”
- “Manage a Configuration Set”

**Introduced before R2006a**
attachConfigSet

Associate configuration set or configuration reference with model

Syntax

attachConfigSet(model, configObj)
attachConfigSet(model, configObj, allowRename)

Arguments

model
The name of an open model, or gcs to specify the current model
configObj
A configuration set (Simulink.ConfigSet) or configuration reference (Simulink.ConfigSetRef)
allowRename
Boolean that determines how Simulink software handles a name conflict

Description

attachConfigSet associates the configuration set or configuration reference (configuration object) specified by configObj with model.

You cannot attach a configuration object to a model if the configuration object is already attached to another model, or has the same name as another configuration object attached to the same model. The optional Boolean argument allowRename determines how Simulink software handles a name conflict between configuration objects. If allowRename is false and the configuration object specified by configObj has the same name as a configuration object already attached to model, Simulink software generates an error. If allowRename is true and a name conflict occurs, Simulink software provides a unique name for configObj before associating configObj with model.
Examples

The following example creates a copy of the current model’s active configuration object and attaches it to the model, changing its name if necessary to be unique. The code is the same whether the object is a configuration set or configuration reference.

```matlab
myConfigObj = getActiveConfigSet(gcs);
copiedConfig = myConfigObj.copy;
copiedConfig.Name = 'DevConfig';
attachConfigSet(gcs, copiedConfig, true);
```

See Also

attachConfigSetCopy | closeDialog | detachConfigSet | getActiveConfigSet | getConfigSet | getConfigSets | openDialog | setActiveConfigSet

Topics

“Manage a Configuration Set”
“Manage a Configuration Reference”

Introduced before R2006a
attachConfigSetCopy

Copy configuration set or configuration reference and associate it with model

Syntax

myConfigObj = attachConfigSetCopy(model, configObj)
myConfigObj = attachConfigSetCopy(model, configObj, allowRename)

Arguments

model
  The name of an open model, or gcs to specify the current model
configObj
  A configuration set (Simulink.ConfigSet) or configuration reference
  (Simulink.ConfigSetRef)
allowRename
  Boolean that specifies how Simulink software handles a name conflict

Description

attachConfigSetCopy copies the configuration set or configuration reference
(configuration object) specified by configObj and associates the copy with model.
Simulink software returns the copied configuration object as newConfigObj.

You cannot attach a configuration object to a model if the configuration object has the
same name as another configuration object attached to the same model. The optional
Boolean argument allowRename determines how Simulink software handles a name
conflict between configuration objects. If allowRename is false and the configuration
object specified by configObj has the same name as a configuration object already
attached to model, Simulink software generates an error. If allowRename is true and a
name conflict occurs, Simulink software provides a unique name for the copy of
configObj before associating it with model.
Examples

The following example creates a copy of ModelA's active configuration object and attaches it to ModelB, changing the name if necessary to be unique. The code is the same whether the object is a configuration set or configuration reference.

```matlab
myConfigObj = getActiveConfigSet('ModelA');
newConfigObj = attachConfigSetCopy('ModelB', myConfigObj, true);
```

See Also

- `attachConfigSet`
- `closeDialog`
- `detachConfigSet`
- `getActiveConfigSet`
- `getConfigSet`
- `getConfigSets`
- `openDialog`
- `setActiveConfigSet`

Topics

- “Manage a Configuration Set”
- “Manage a Configuration Reference”

Introduced in R2006b
addLabel

(Not recommended) Attach label to project file

**Note**  `simulinkproject` is not recommended. Use `currentProject` or `openProject` instead. For more information, see “Compatibility Considerations”.

**Syntax**

`addLabel(file, categoryName, labelName)`

`addLabel(file, categoryName, labelName, labelData)`

**Description**

`addLabel(file, categoryName, labelName)` attaches the specified label `labelName` in the category `categoryName` to the file.

`addLabel(file, categoryName, labelName, labelData)` attaches the label with data `labelData`.

**Examples**

**Attach a Label to a Project File**

Open the airframe project and create a project object.

```matlab
sldemo_slproject_airframe;
proj = simulinkproject;
```

Get a particular file by name.

```matlab
myfile = findFile(proj, 'models/AnalogControl.slx')
```

`myfile =`
ProjectFile with properties:

    Path: 'C:\Work\temp\slexamples\airframe\models\AnalogControl.slx'
    Labels: [1x1 slproject.Label]
    Revision: '2'
    SourceControlStatus: Unmodified

Get the Labels property of the file.

myfile.Labels

ans =

    Label with properties:

        File: 'C:\Work\temp\slexamples\airframe\models\AnalogControl.slx'
        Data: []
        DataType: 'none'
        Name: 'Design'
        CategoryName: 'Classification'

Attach the label 'Artifact' to the file.

addLabel(myfile,'Classification','Artifact')

ans =

    Label with properties:

        File: 'C:\Work\temp\slexamples\airframe\models\AnalogControl.slx'
        Data: []
        DataType: 'none'
        Name: 'Artifact'
        CategoryName: 'Classification'

Index into the Labels property to get the label attached to this file.

reviewlabel = myfile.Labels(1)

reviewlabel =

    Label with properties:

        File: 'C:\Work\temp\slexamples\airframe\models\AnalogControl.slx'
        Data: []
        DataType: 'none'
Name: 'Artifact'
CategoryName: 'Classification'

Detach the new label from the file.
removeLabel(myfile, reviewlabel)

### Attach a Label to a Subset of Files

Attach the 'Classification' category label 'Utility' to all files in the project that have the .m file extension.

Open the airframe project and create a project object.

```matlab
sldemo_slproject_airframe;
proj = simulinkproject;
```

Get the list of files.

```matlab
files = proj.Files;
```

Loop through each file. If a file has the extension .m, attach the label 'Utility'.

```matlab
for fileIndex = 1:numel(files)
    file = files(fileIndex);
    [~, ~, fileExtension] = fileparts(file.Path);
    if strcmp(fileExtension, '.m')
        addLabel(file, 'Classification', 'Utility');
    end
end
```

In the Project Files view, the Classification column displays the label Utility for each .m file in the utilities folder.

### Attach a Label and Label Data to a File

Open the airframe project and create a project object.

```matlab
sldemo_slproject_airframe;
proj = simulinkproject;
```

Create a new category 'Review'.

2-34
createCategory(proj,'Review','char');

For the new category, create a label 'To Review'.

reviewCategory = findCategory(proj,'Review');
cREATE LABEL(reviewCategory,'To Review');

Get a particular file by name.

myfile = findFile(proj,'models/AnalogControl.slx')

myfile =

ProjectFile with properties:
    Path: 'C:\Work\temp\slexamples\airframe\models\AnalogControl.slx'
    Labels: [1x1 slproject.Label]
    Revision: '2'
    SourceControlStatus: Unmodified

Attach the label 'To Review' and a character vector of label data to the file.

addLabel(myfile,'Review','To Review','Whole team design review')

Index into the Labels property to get the second label attached to this particular file, and see the label data.

myfile.Labels(2)

ans =

Label with properties:
    File: 'C:\Work\temp\slexamples\airframe\models\AnalogControl.slx'
    Data: 'Whole team design review'
    DataType: 'char'
    Name: 'To Review'
    CategoryName: 'Review'

In the Project Files view, for the AnalogControl.slx file, the Review column displays the To Review label with label data.

Alternatively, you can set or change label data using the data property.
mylabel = myfile.Labels(2);
mylabel.Data = 'Final review';

**Input Arguments**

**file** — File to attach label to
file object

File to attach the label to, specified as a file object. You can get the file object by examining the project’s Files property (proj.Files), or use findFile to find a file by name. The file must be in the project.

**categoryName** — Name of category for label
character vector

Name of the category for the label, specified as a character vector.

**labelName** — Name of label
character vector | label definition object

Name of the label to attach, specified as a character vector or a label definition object returned by the file.Label property or findLabel. You can specify a new label name that does not already exist in the project.

**labelData** — Data to attach to label
character vector | numeric

Data to attach to the label, specified as a character vector or numeric. Data type depends on the label definition. Get a label to examine its DataType property using file.Label or findLabel.

**Compatibility Considerations**

**Simulink project API is not recommended**

*Not recommended starting in R2019a*

Starting in R2019a, instead of simulinkproject and related functions, use the currentProject or openProject functions in MATLAB. The Simulink project API will

2-36
continue to be supported, but, after R2019a, new features might be available only if you use the new MATLAB project API. There are no plans to remove the Simulink project API at this time.

The new MATLAB project API is part of the R2019a functionality enabling you to use projects in MATLAB, with or without Simulink. You can now share projects with users who do not have Simulink.

See Also

Functions
createLabel | findFile | findLabel | removeLabel | simulinkproject

Introduced in R2013a
batchsim

Offload simulations to run on a compute cluster

Syntax

simJob = batchsim(in)
simJob = batchsim(myCluster,in)
simJob = batchsim(...,Name,Value)

Description

simJob = batchsim(in) runs a batch job on a single worker to simulate a model using the inputs specified in the SimulationInput object, in.

simJob = batchsim(myCluster,in) runs a batch job on the cluster identified by the cluster object myCluster. If a cluster profile is not specified, batchsim uses a default cluster profile as set up in the parallel preferences. For more information, see “Discover Clusters and Use Cluster Profiles” (Parallel Computing Toolbox).

simJob = batchsim(...,Name,Value) runs a batch job that simulates a model using the inputs specified in the SimulationInput object and the options specified as Name,Value pair.

batchsim offloads simulations to a compute cluster, enabling you to carry out other tasks while the batch job is processing, or close the client MATLAB and access the batch job later. Use the 'Pool' argument to run simulations in parallel.

The batchsim command uses the Parallel Computing Toolbox™ and MATLAB Parallel Server™ licenses to run the simulations on compute cluster. batchsim runs the simulations in serial if a parallel pool cannot be created. If Parallel Computing Toolbox license is not available, batchsim errors out.

Examples
Run Parallel Simulations with `batchsim`

This example shows how to run parallel simulations in batch. `batchsim` offloads simulations to a compute cluster, enabling you to carry out other tasks while the batch job is processing, or close the client MATLAB and access the batch job later.

This example uses the `ex_sldemo_househeat` model and runs simulations in batch to observe the model behavior for different temperature set points.

Open the model.

```matlab
openExample('simulink/OpenTheModelExample');
open_system('ex_sldemo_househeat');
load_system('ex_sldemo_househeat')
```

Define a set of values for different temperatures.

```matlab
setPointValues = 65:2:85;
spv_Length = length(setPointValues);
```

Using the `setPointValues`, initialize an array of `Simulink.SimulationInput` objects.

```matlab
in(1:spv_Length) = Simulink.SimulationInput('ex_sldemo_househeat');
for i = 1:1:spv_Length
    in(i) = in(i).setBlockParameter('ex_sldemo_househeat/Set Point','Value',num2str(setPointValues(i)));
end
```

Specify the pool size of the number of workers to use. In addition to the number of workers used to run simulations in parallel, a head worker is required. In this case, let's assume that three workers are available to run a batch job for the parallel simulations.

The job object returns useful metadata as shown. You can use the job ID to access the job object later from any machine. `NumWorkers` tells you how many workers are running the simulations. `NumWorkers` is always the number of workers specified in the 'Pool' argument and an additional head worker.

```matlab
simJob = batchsim(in,'Pool',3)
```

```
ID: 1
Type: pool
NumWorkers: 4
Username: #####
State: running
```
Access the results of the batch job using the `fetchOutputs` method. `fetchOutputs` returns an array of `Simulink.SimulationOutput` objects.

```matlab
out = fetchOutputs(simJob)
```

**1x11 Simulink.SimulationOutput array**

### Input Arguments

**`in` — Simulink.SimulationInput object used to simulate the model**

Object, array

Specified as a `Simulink.SimulationInput` object or an array of `Simulink.SimulationInput` objects that is used to specify changes to the model for simulation.

Example: `in = Simulink.SimulationInput('vdp')`

**myCluster — `parallel.Cluster` object**

Object

Cluster object that is used to specify the cluster in which the batch job runs.

### Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'Pool', 5`

**AdditionalPaths — Files to attach to parallel pool**

Character vector | cell array

Specified as a character vector, a cell array or an array of character vector to define paths to be added to the MATLAB search path of the workers before the simulations execute. The default search path might not be the same on the workers as it is on the client; the
path difference could be the result of different current working folders (pwd), platforms, or network file system access. The 'AdditionalPaths' property can assure that workers are looking in the correct locations for necessary code files, data files, model files, etc.

**AttachedFiles — Files to attach to parallel pool**
cell array

Specified as a cell array of additional files to attach to the parallel pool.

**AutoAddClientPath — Whether user-added entries on client path are added to each worker path**
true (default) | false

Specified as true or false to control whether user-added entries on the client path are added to each worker path.

**AutoAttachedFiles — Whether code files should be automatically attached to the job**
true (default) | false

Specified as true or false to control whether code files are automatically attached to the job.

**CaptureDiary — Whether diary is collected**
true (default) | false

Specified as true or false to indicate collection of the diary.

**CleanupFcn — Function handle to run once per worker after running simulations**
function handle

Specify a function handle to 'CleanupFcn' to run once per worker after the simulations are completed.

**EnvironmentVariables — Names of environment variables copied from client session to workers**
character vector | cell array

Specifies the names of environment variables copied from the client session to the workers. The names specified here are appended to the 'EnvironmentVariables' property specified in the applicable parallel profile to form the complete list of environment variables. Any variables listed which are not set are not copied to the
workers. These environment variables will be set on the workers for the duration of the batch job.

**ManageDependencies — Manage model dependencies**

'off' (default) | 'on'

When ManageDependencies is set to 'on', model dependencies are automatically sent to the parallel workers if necessary. If ManageDependencies is set to 'off', explicitly attach model dependencies to the parallel pool.

**Pool — Size of the number of workers for a parallel pool**

An integer specifying the number of workers to make into a parallel pool for the job in addition to the worker running the batch job itself. The simulations use this pool for execution. Because the pool requires N workers in addition to the worker running the batch, there must be at least N+1 workers available on the cluster.

**Profile — Cluster profile name**

The name of a cluster profile used to identify the cluster. If this option is omitted, the default profile is used to identify the cluster and is applied to the job and task properties.

**SetupFcn — Function handle to run once per worker**

Specify a function handle to 'SetupFcn' to run once per worker before the start of the simulations.

**Note** When `buildRapidAcceleratorTarget` is used in the `SetupFcn` and the model has external inputs specified, either set 'LoadExternalInput' to 'off' or ensure that the specified external input is available on the workers to prevent compilation error.

**ShowProgress — Show the progress of the simulations in diary**

'off' | 'on'

Set to 'on', to copy the progress of the simulations in the command window to diary of `Simulink.Simulation.Job` object. The progress is hidden when set to 'off'.

2-42
**StopOnError — Stop simulations on errors**
'off' (default) | 'on'

Setting 'StopOnError' to 'on' stops the execution of simulations if an error is encountered.

**TransferBaseWorkspaceVariables — Transfer variables to the parallel workers**
'off' (default) | 'on'

When TransferBaseWorkspaceVariables is set to true, variables used in the model and defined in the base workspace are transferred to the parallel workers.

**Note** Use of TransferBaseWorkspaceVariables requires model compilation.

**UseFastRestart — Use fast restart**
'off' (default) | 'on'

When UseFastRestart is set to true, simulations run on the workers using fast restart.

**Note** When using batchsim, use the UseFastRestart option and not the FastRestart option. See “Get Started with Fast Restart” for more information.

**Output Arguments**

**simJob — Simulink.Simulation.Job job object**

object

An object containing metadata of submitted batch job. Poll job object using it's ID to check the status of simulations or to access outputs on completion of the job.
Extended Capabilities

Automatic Parallel Support
Accelerate code by automatically running computation in parallel using Parallel Computing Toolbox™.

Using `batchsim` with Parallel Computing Toolbox installed, MATLAB automatically opens a worker and runs the job in the background on another session on your local machine. Specifying a pool size runs the simulations on the number of workers specified. Control parallel behavior with the parallel preferences, including scaling up to a cluster.

For details, see “Run Multiple Simulations”.

See Also

Functions
`batch` | `batchsim` | `cancel` | `diary` | `fetchOutputs` | `getSimulationJobs` | `listAutoAttachedFiles` | `parcluster` | `parsim` | `wait`

Classes
`Simulink.Simulation.Job` | `Simulink.SimulationInput`

Topics
“Multiple Simulation Workflows”
“Run Multiple Simulations”
“Batch Processing” (Parallel Computing Toolbox)
“Job Monitor” (Parallel Computing Toolbox)

Introduced in R2018b
bdclose

Close any or all Simulink system windows unconditionally

Syntax

bdclose
bdclose(sys)
bdclose('all')

Description

bdclose closes the current system window unconditionally and without confirmation. Any changes made to the system since it was last saved are lost. To find out the current system, use gcs. If a model was only loaded, bdclose clears the model from memory.

bdclose(sys) closes the specified system window, discarding all changes.

bdclose('all') closes all open system windows, discarding all changes.

Examples

Close a System Without Saving

Open the vdp system. Then close it without saving any changes.

open_system(vdp)
bdclose('vdp')

Input Arguments

sys — System to close
character vector | string scalar
System to close, specified as a character vector or string scalar.
Example: "vdp"

See Also
close_system|new_system|open_system|save_system

Introduced before R2006a
**bdIsDirty**

Whether block diagram has unsaved changes

**Syntax**

\[ \text{isDirty} = \text{bdIsDirty}(\text{bdname}) \]

**Description**

\[ \text{isDirty} = \text{bdIsDirty}(\text{bdname}) \]

returns whether or not the loaded block diagram \text{bdname} has unsaved changes.

**Examples**

**Check Models for Unsaved Changes**

Check if models contain unsaved changes using \text{bdIsDirty}.

Check if a single model is dirty.

\begin{verbatim}
vdp
bdIsDirty('vdp')
\end{verbatim}

\begin{verbatim}
ans =
   logical
    0
\end{verbatim}

Check if a cell array of models are dirty.

\begin{verbatim}
vdp
sf_car
bdIsDirty({'sf_car','vdp'})
\end{verbatim}
ans =
1×2 logical array
0 0

Input Arguments

bdname — Loaded block diagram name
character vector | cell array of character vectors | double array

Loaded block diagram name, specified as a character vector, a cell array of character vectors, or a double array. All character vectors must be the names of loaded block diagrams. All doubles must be the handles of loaded block diagrams. It is an error to supply an invalid handle, a handle to anything other than a block diagram, a path to a block or subsystem, or a block diagram that is not loaded.

Data Types: double | char | cell

Output Arguments

isDirty — Whether block diagram has unsaved changes
logical scalar | logical array

Whether block diagram has unsaved changes, returned as a logical array with one entry for each block diagram. The logical value is true if the block diagram has been modified in memory since it was loaded or last saved, and false if there are no unsaved changes.

See Also

bdIsLoaded

Topics
“Manage Shadowed and Dirty Models and Other Project Files”

Introduced in R2017a
**bdIsLibrary**

Whether block diagram is a library

**Syntax**

\[
\text{isLibrary} = \text{bdIsLibrary}(\text{bdnames})
\]

**Description**

\(\text{isLibrary} = \text{bdIsLibrary}(\text{bdnames})\) returns whether the loaded block diagrams specified by \(\text{bdnames}\) are libraries.

**Examples**

**Check Whether Block Diagrams Are Libraries**

Load some block diagrams and get a handle to one of them.

```
load_system({'sf_car','hydlib','vdp'})
h = get_param('hydlib','Handle');
```

Check whether \(\text{sf_car}\) is a library. The returned value 0 indicates that it is not.

```
bdIsLibrary('sf_car')
ans =
 0
```

Check whether \(\text{hydlib}\) and \(\text{vdp}\) are libraries. The returned value shows that \(\text{hydlib}\) is a library and \(\text{vdp}\) is not.

```
bdIsLibrary({'hydlib','vdp'})
ans =
1 0
```
Using the handle to hydlib, check whether hdlib is a library. The value returned shows that it is.

bdIsLibrary(h)

ans =
1

Input Arguments

bdnames — Names or handles of loaded block diagrams
character vector | cell array of character vectors | double | array of doubles

Names or handles of loaded block diagrams, specified as a character vector, a cell array of character vectors, a double, or a double array. All character vectors are names of loaded block diagrams. All doubles are handles of loaded block diagrams.

Data Types: char | cell | double

Output Arguments

isLibrary — Logical array showing whether block diagrams are libraries
logical scalar | logical array

Logical array showing whether block diagrams are libraries, returned as a logical scalar or array (1 for a library, 0 otherwise).

See Also

bdIsLoaded | bdroot | find_system

Introduced in R2015a
bdIsLoaded

Whether block diagram is in memory

Syntax

isLoaded = bdIsLoaded(bdnames)

Description

isLoaded = bdIsLoaded(bdnames) returns whether or not a block diagram is in memory. bdnames can be a character vector or a cell array of character vectors. All character vectors must be valid block diagram names. It is an error to supply a path to a block or subsystem.

isLoaded is a logical array with one entry for each block diagram name.

Examples

bdIsLoaded('sf_car')

returns a logical scalar.

bdIsLoaded({'sf_car','vdp'})

returns a 1-by-2 logical array.

See Also

bdIsDirty | bdIsLibrary | find_system

Introduced in R2008a
**bdroot**

Top-level model of current system

**Syntax**

model = bdroot
model = bdroot(elements)

**Description**

model = bdroot returns the top-level model of the current system. The current system is the currently active Simulink Editor window or the system in which a block is selected.

model = bdroot(elements) returns the top-level model of the specified model elements. Before using bdroot, make sure the top-level model of each element in elements is loaded.

**Examples**

**Get Top-Level Model of Current System**

Open the system Controller in the model f14.

load_system('f14')
open_system('f14/Controller')

Get the top-level model of the current system.

bdroot

ans =

'f14'
Get Top-Level Model of a System

Open the system Controller in the model f14.

```matlab
load_system('f14')
open_system('f14/Controller')
```

Get the top-level model of the current system.

```matlab
bdroot(gcs)
```

```
ans =
    'f14'
```

Input Arguments

elements — Model elements whose top-level models to return

*model name | block path name | handle | cell array of character vectors | string array | numeric array*

Model elements whose top-level model to return, specified as the model name, block or system path name, handle, cell array of character vectors or string array of system names, or numeric array of handles.

**Tip** Use `bdroot` with `gcs`, `gcb`, and `gcbh` to get the top-level model of the current system or block.

Output Arguments

**model — Top-level model**

*character vector | cell array | string array*

Top-level model, returned as a character vector of the model name. If the input was an array, `model` is returned as an array of the same type as the input.

See Also

gcb | gcbh | gcs
Introduced before R2006a
dlinmod

Extract discrete-time linear state-space model around operating point

Syntax

argout = dlinmod('sys', Ts)
argout = dlinmod('sys', Ts, x, u)
argout = dlinmod('sys', Ts, x, u, para, 'v5')
argout = dlinmod('sys', Ts, x, u, para, xpert, upert, 'v5')

Arguments

sys Name of the Simulink system from which the linear model is extracted.

x, u State (x) and the input (u) vectors. If specified, they set the operating point at which the linear model is extracted. When a model has model references using the Model block, you must use the Simulink structure format to specify x. To extract the x structure from the model, use the following command:

```
x = Simulink.BlockDiagram.getInitialState('sys');
```

You can then change the operating point values within this structure by editing x.signals.values.

If the state contains different data types (for example, 'double' and 'uint8'), then you cannot use a vector to specify this state. You must use a structure instead. In addition, you can only specify the state as a vector if the state data type is 'double'.

Ts Sample time of the discrete-time linearized model
An optional argument that invokes the perturbation algorithm created prior to MATLAB 5.3. Invoking this optional argument is equivalent to calling `linmodv5`.

*para* A three-element vector of optional arguments:

- `para(1)` — Perturbation value of delta, the value used to perform the perturbation of the states and the inputs of the model. This is valid for linearizations using the 'v5' flag. The default value is 1e-05.

- `para(2)` — Linearization time. For blocks that are functions of time, you can set this parameter with a nonnegative value that gives the time (t) at which Simulink evaluates the blocks when linearizing a model. The default value is 0.

- `para(3)` — Set `para(3)=1` to remove extra states associated with blocks that have no path from input to output. The default value is 0.

*xpert, upert* The perturbation values used to perform the perturbation of all the states and inputs of the model. The default values are

\[
\begin{align*}
xpert &= para(1) + 1e-3 * para(1) * abs(x) \\
upert &= para(1) + 1e-3 * para(1) * abs(u)
\end{align*}
\]

When a model has model references using the Model block, you must use the Simulink structure format to specify *xpert*. To extract the *xpert* structure, use the following command:

\[
xpert = Simulink.BlockDiagram.getInitialState('sys');
\]

You can then change the perturbation values within this structure by editing *xpert*.signals.values.

The perturbation input arguments are only available when invoking the perturbation algorithm created prior to MATLAB 5.3, either by calling `linmodv5` or specifying the 'v5' input argument to `linmod`. 
argout  
linmod, dlinmod, and linmod2 return state-space representations if you specify the output (left-hand) side of the equation as follows:

- \([A,B,C,D] = \text{linmod}('sys', x, u)\) obtains the linearized model of \(sys\) around an operating point with the specified state variables \(x\) and the input \(u\). If you omit \(x\) and \(u\), the default values are zero.

linmod and dlinmod both also return a transfer function and MATLAB data structure representations of the linearized system, depending on how you specify the output (left-hand) side of the equation. Using linmod as an example:

- \([\text{num}, \text{den}] = \text{linmod}('sys', x, u)\) returns the linearized model in transfer function form.
- \(\text{sys} \_\text{struc} = \text{linmod}('sys', x, u)\) returns a structure that contains the linearized model, including state names, input and output names, and information about the operating point.

**Description**

dlinmod compute a linear state-space model for a discrete-time system by linearizing each block in a model individually.

linmod obtains linear models from systems of ordinary differential equations described as Simulink models. Inputs and outputs are denoted in Simulink block diagrams using Inport and Outport blocks.

The default algorithm uses preprogrammed analytic block Jacobians for most blocks which should result in more accurate linearization than numerical perturbation of block inputs and states. A list of blocks that have preprogrammed analytic Jacobians is available in the Simulink Control Design documentation along with a discussion of the block-by-block analytic algorithm for linearization.

The default algorithm also allows for special treatment of problematic blocks such as the Transport Delay and the Quantizer. See the mask dialog of these blocks for more information and options.
Discrete-Time System Linearization

The function `dlinmod` can linearize discrete, multirate, and hybrid continuous and discrete systems at any given sampling time. Use the same calling syntax for `dlinmod` as for `linmod`, but insert the sample time at which to perform the linearization as the second argument. For example,

```
[Ad,Bd,Cd,Dd] = dlinmod('sys', Ts, x, u);
```

produces a discrete state-space model at the sampling time `Ts` and the operating point given by the state vector `x` and input vector `u`. To obtain a continuous model approximation of a discrete system, set `Ts` to 0.

For systems composed of linear, multirate, discrete, and continuous blocks, `dlinmod` produces linear models having identical frequency and time responses (for constant inputs) at the converted sampling time `Ts`, provided that

- `Ts` is an integer multiple of all the sampling times in the system.
- The system is stable.

For systems that do not meet the first condition, in general the linearization is a time-varying system, which cannot be represented with the `[A,B,C,D]` state-space model that `dlinmod` returns.

Computing the eigenvalues of the linearized matrix `Ad` provides an indication of the stability of the system. The system is stable if `Ts>0` and the eigenvalues are within the unit circle, as determined by this statement:

```
all(abs(eig(Ad))) < 1
```

Likewise, the system is stable if `Ts = 0` and the eigenvalues are in the left half plane, as determined by this statement:

```
all(real(eig(Ad))) < 0
```

When the system is unstable and the sample time is not an integer multiple of the other sampling times, `dlinmod` produces `Ad` and `Bd` matrices, which can be complex. The eigenvalues of the `Ad` matrix in this case still, however, provide a good indication of stability.

You can use `dlinmod` to convert the sample times of a system to other values or to convert a linear discrete system to a continuous system or vice versa.
You can find the frequency response of a continuous or discrete system by using the `bode` command.

**Notes**

By default, the system time is set to zero. For systems that are dependent on time, you can set the variable `para` to a two-element vector, where the second element is used to set the value of `t` at which to obtain the linear model.

The ordering of the states from the nonlinear model to the linear model is maintained. For Simulink systems, a character vector variable that contains the block name associated with each state can be obtained using

```
[sizes,x0,xstring] = sys
```

where `xstring` is a vector of strings whose `i`th row is the block name associated with the `i`th state. Inputs and outputs are numbered sequentially on the diagram.

For single-input multi-output systems, you can convert to transfer function form using the routine `ss2tf` or to zero-pole form using `ss2zp`. You can also convert the linearized models to LTI objects using `ss`. This function produces an LTI object in state-space form that can be further converted to transfer function or zero-pole-gain form using `tf` or `zpk`.

The default algorithms in `linmod` and `dlinmod` handle Transport Delay blocks by replacing the linearization of the blocks with a Pade approximation. For the `v5` algorithm, linearization of a model that contains Derivative or Transport Delay blocks can be troublesome. For more information, see “Linearizing Models”.

**See Also**

`linmod` | `linmod2` | `linmodv5`

**Introduced in R2007a**
close_system

Close Simulink system window or block dialog box

Syntax

close_system
close_system(blockOrSys)
close_system(sys,saveflag)
close_system(sys,newsys)
close_system(sys,newsys,Name,Value)

Description

close_system closes the current system or subsystem. If the current system is the top-level model and has been modified, close_system returns an error.

Use the gcs function to determine the current system.

**Note** You cannot use close_system in a block or menu callback to close the root-level model. Attempting to close the root-level model in a block or menu callback causes an error and discontinues the callback execution.

close_system(blockOrSys) closes the specified model, subsystem, or the block dialog box of the specified block. If the model was only loaded, that is, with load_system, this syntax clears the model from memory.

close_system(sys,saveflag) lets you specify whether to save the model with its current name or to close without saving.

close_system(sys,newsys) saves the model to a file with the specified name before closing.

close_system(sys,newsys,Name,Value) closes and saves the model with additional options specified by one or more Name,Value pair arguments.
Examples

Close the Current System

Open the model vdp and then close it.

vdp
close_system

Close a Block Dialog Box

Open the model vdp.

vdp

In the model, to open its block dialog box, double-click the block Mu.

Close the block dialog box

close_system('vdp/Mu');

Close a Model and Save with New Name

Open a model, modify it, save the model with a new name and close it.

Open the model vdp and add a block to it.

vdp
block = add_block('vdp/Mu','vdp/Mu','MakeNameUnique','on');

Close the model, saving it with a new name.

close_system('vdp','myvdp');

Close a Model Using Name,Value Options

This command tries to save the vdp system to a file with the name 'max', but returns an error because 'max' is the name of an existing MATLAB function.
close_system('vdp','max','ErrorIfShadowed',true)

**Input Arguments**

`blockOrSys` — Block, model, or subsystem to close  
character vector | cell array of character vectors | string array | handle | array of handles

Name of model, subsystem, or the block whose dialog box you want to close, specified as a character vector, cell array of character vectors, string array, handle, or array of handles. Do not use a file extension.

Example: "vdp/Mu" 'vdp'

`sys` — Name of model to close  
character vector | cell array of character vectors | string array | handle | array of handles

Name of model to close, specified as a character vector, cell array of character vectors, string array, handle, or array of handles. Do not use a file extension.

`saveflag` — Option to save file with current name  
0 (default) | 1 | numeric array

Option to save model using the current file name, specified as 0 to close without saving or 1 to save and then close. If `sys` is an array, you can supply a single character to apply to all of the models in the array. Or you can provide a numeric array with values that correspond with each model in the `sys` array.

For information on rules for naming models, see “Model Names”.

`newsys` — File to save to  
character vector | cell array of character vectors | string array

File to save to, specified as a character vector, cell array of character vectors, or string array. You can specify a model name in the current folder or the full path name, with or without an extension.

With no an extension, `close_system` saves to the file format specified in your Simulink preferences. Possible model extensions are `.slx` and `.mdl`.

For information on rules for naming models, see “Model Names”.
Name-Value Pair Arguments

Specify optional comma-separated pairs of Name, Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside quotes. You can specify several name and value pair arguments in any order as Name1, Value1, ..., NameN, ValueN.

Example:
save_system('mymodel','newmodel','SaveModelWorkspace',true,'BreakUse rLinks',true,'OverwriteIfChangedOnDisk',true)

**ErrorIfShadowed — Return an error if name exists**
false (default) | true | 'on' | 'off'

Option to return an error if the new name exists on the MATLAB path or workspace, specified as true, false, 'on', or 'off'.

**OverwriteIfChangedOnDisk — Overwrite file**
false (default) | true | 'on' | 'off'

Option to overwrite the file on disk even if it has been modified since the system was loaded, specified as true, false, 'on', or 'off'. By default, if the file changed on disk since the model was loaded, close_system displays an error to prevent the changes on disk from being overwritten.

If the file has changed on disk, you can control whether save_system displays an error using a Simulink preference. In the Model File pane of the Simulink Preferences dialog box, under Change Notification, select Saving the model. This preference is on by default.

**SaveModelWorkspace — Save model workspace**
false (default) | true | 'on' | 'off'

Option to save the contents of the model workspace, specified as true, false, 'on', or 'off'. The model workspace DataSource must be a MAT-file. If the data source is not a MAT-file, save_system does not save the workspace. See “Specify Source for Data in Model Workspace”.

**See Also**
bdclose | gcs | load_system | new_system | open_system | save_system
Introduced before R2006a
**closeDialog**

Close configuration parameters dialog

**Syntax**

closeDialog(configObj)

**Arguments**

*configObj*

A configuration set (`Simulink.ConfigSet`) or configuration reference (`Simulink.ConfigSetRef`)

**Description**

closeDialog closes an open configuration parameters dialog box. If `configObj` is a configuration set, the function closes the dialog box that displays the configuration set. If `configObj` is a configuration reference, the function closes the dialog box that displays the referenced configuration set, or generates an error if the reference does not specify a valid configuration set. If the dialog box is already closed, the function does nothing.

**Examples**

The following example closes a configuration parameters dialog box that shows the current parameters for the current model. The parameter values derive from the active configuration set or configuration reference (configuration object). The code is the same in either case; the only difference is which type of configuration object is currently active.

```matlab
myConfigObj = getActiveConfigSet(gcs);
closeDialog(myConfigObj);
```
See Also
attachConfigSet | attachConfigSetCopy | detachConfigSet |
getActiveConfigSet | getConfigSet | getConfigSets | openDialog |
setActiveConfigSet

Topics
“Manage a Configuration Set”
“Manage a Configuration Reference”

Introduced in R2006b
close

(Not recommended) Close project

**Note** `simulinkproject` is not recommended. Use `currentProject` or `openProject` instead of `simulinkproject`. For more information, see “Compatibility Considerations”.

**Syntax**

close(proj)

**Description**

close(proj) closes the project `proj`.

**Examples**

**Open and Close a Project**

Open a specified project and get a project object to manipulate the project at the command line. For example,

```matlab
proj = simulinkproject('C:/projects/project1/myproject.prj')
```

Close the project.

close(proj)

**Input Arguments**

`proj` — Project

project object
Project, specified as a project object. Use `simulinkproject` to create a project object to manipulate a project at the command line.

Compatibility Considerations

**Simulink project API is not recommended**

*Not recommended starting in R2019a*

Starting in R2019a, instead of `simulinkproject` and related functions, use the `currentProject` or `openProject` functions in MATLAB. The Simulink project API will continue to be supported, but, after R2019a, new features might be available only if you use the new MATLAB project API. There are no plans to remove the Simulink project API at this time.

The new MATLAB project API is part of the R2019a functionality enabling you to use projects in MATLAB, with or without Simulink. You can now share projects with users who do not have Simulink.

See Also

**Functions**

`simulinkproject`

**Introduced in R2013a**
coder.allowpcode

Package: coder

Control code generation from protected MATLAB files

Syntax

coder.allowpcode('plain')

Description

coder.allowpcode('plain') allows you to generate protected MATLAB code (P-code) that you can then compile into optimized MEX functions or embeddable C/C++ code. This function does not obfuscate the generated MEX functions or embeddable C/C++ code.

With this capability, you can distribute algorithms as protected P-files that provide code generation optimizations, providing intellectual property protection for your source MATLAB code.

Call this function in the top-level function before control-flow statements, such as if, while, switch, and function calls.

MATLAB functions can call P-code. When the .m and .p versions of a file exist in the same folder, the P-file takes precedence.

coder.allowpcode is ignored outside of code generation.

Examples

Generate optimized embeddable code from protected MATLAB code:

1. Write an function p_abs that returns the absolute value of its input:

   function out = p_abs(in)   %#codegen
   % The directive %#codegen indicates that the function
   % The code generation directives are used as comments in the code generation process.
% is intended for code generation
coder.allowpcode('plain');
out = abs(in);

2 Generate protected P-code. At the MATLAB prompt, enter:

```
pcode p_abs
```

The P-file, `p_abs.p`, appears in the current folder.

3 Generate a MEX function for `p_abs.p`, using the `-args` option to specify the size, class, and complexity of the input parameter (requires a MATLAB Coder license). At the MATLAB prompt, enter:

```
codegen p_abs -args { int32(0) }
```

codegen generates a MEX function in the current folder.

4 Generate embeddable C code for `p_abs.p` (requires a MATLAB Coder license). At the MATLAB prompt, enter:

```
codegen p_abs -config:lib -args { int32(0) };
```

codegen generates C library code in the `codegen\lib\p_abs` folder.

See Also

codegen | pcode

Introduced in R2011a
coder.ceval

Call external C/C++ function

Syntax

coder.ceval(cfun_name)
coder.ceval(cfun_name,cfun_arguments)

coder.ceval('-global',cfun_name)
coder.ceval('-global',cfun_name,cfun_arguments)

coder.ceval('-layout:rowMajor',cfun_name,cfun_arguments)
coder.ceval('-layout:columnMajor',cfun_name,cfun_arguments)
coder.ceval('-layout:any',cfun_name,cfun_arguments)

cfun_return = coder.ceval(__)

Description

coder.ceval(cfun_name) executes the external C/C++ function specified by cfun_name. Define cfun_name in an external C/C++ source file or library. Provide the external source, library, and header files to the code generator.

coder.ceval(cfun_name,cfun_arguments) executes cfun_name with arguments cfun_arguments. cfun_arguments is a comma-separated list of input arguments in the order that cfun_name requires.

By default, coder.ceval passes arguments by value to the C/C++ function whenever C/C++ supports passing arguments by value. To make coder.ceval pass arguments by reference, use the constructs coder.ref, coder.rref, and coder.wref. If C/C++ does not support passing arguments by value, for example, if the argument is an array, coder.ceval passes arguments by reference. If you do not use coder.ref, coder.rref or coder.wref, a copy of the argument can appear in the generated code to enforce MATLAB semantics for arrays.
coder.ceval('-global', cfun_name) executes cfun_name and indicates that cfun_name uses one or more MATLAB global variables. The code generator can then produce code that is consistent with this global variable usage.

coder.ceval('-global', cfun_name, cfun_arguments) executes cfun_name with arguments cfun_arguments and indicates that cfun_name uses one or more MATLAB global variables.

coder.ceval('-layout:rowMajor', cfun_name, cfun_arguments) executes cfun_name with arguments cfun_arguments and passes data stored in row-major layout. When called from a function that uses column-major layout, the code generator converts inputs to row-major layout and converts outputs back to column-major layout. For a shorter syntax, use coder.ceval('-row', ...).

coder.ceval('-layout:columnMajor', cfun_name, cfun_arguments) executes cfun_name with arguments cfun_arguments and passes data stored in column-major layout. When called from a function that uses row-major layout, the code generator converts inputs to column-major layout and converts outputs back to row-major layout. For a shorter syntax, use coder.ceval('-col', ...).

coder.ceval('-layout:any', cfun_name, cfun_arguments) executes cfun_name with arguments cfun_arguments and passes data with its current array layout, even when array layouts do not match. The code generator does not convert the array layout of the input or output data.

cfun_return = coder.ceval(____) executes cfun_name and returns a single scalar value, cfun_return, corresponding to the value that the C/C++ function returns in the return statement. To be consistent with C/C++, coder.ceval can return only a scalar value. It cannot return an array. Use this option with any of the input argument combinations in the previous syntaxes.

Examples

Call External C Function

Call a C function foo(u) from a MATLAB function from which you intend to generate C code.

Create a C header file foo.h for a function foo that takes two input parameters of type double and returns a value of type double.
double foo(double in1, double in2);

Write the C function foo.c.

#include <stdio.h>
#include <stdlib.h>
#include "foo.h"

double foo(double in1, double in2)
{
    return in1 + in2;
}

Write a function callfoo that calls foo by using coder.ceval. Provide the source and header files to the code generator in the function.

function y = callfoo  %#codegen
y = 0.0;
if coder.target('MATLAB')
    % Executing in MATLAB, call MATLAB equivalent of
    % C function foo
    y = 10 + 20;
else
    % Executing in generated code, call C function foo
    coder.updateBuildInfo('addSourceFiles','foo.c');
    coder.cinclude('foo.h');
    y = coder.ceval('foo', 10, 20);
end
end

Generate C library code for function callfoo. The codegen function generates C code in the \codegen\lib\callfoo subfolder.

codegen -config:lib callfoo -report

Call a C Library Function

Call a C library function from MATLAB code.

Write a MATLAB function myabsval.
function y = myabsval(u)
    %#codegen
    y = abs(u);

Generate a C static library for `myabsval`, using the `-args` option to specify the size, type, and complexity of the input parameter.

codegen -config:lib myabsval -args {0.0}

The `codegen` function creates the library file `myabsval.lib` and header file `myabsval.h` in the folder `codegen/lib/myabsval`. (The library file extension can change depending on your platform.) It generates the functions `myabsval_initialize` and `myabsval_terminate` in the same folder.

Write a MATLAB function to call the generated C library function using `coder.ceval`.

function y = callmyabsval(y)
    %#codegen
    % Check the target. Do not use `coder.ceval` if `callmyabsval` is
    % executing in MATLAB
    if coder.target('MATLAB')
        % Executing in MATLAB, call function `myabsval`
        y = myabsval(y);
    else
        % add the required include statements to generated function code
        coder.updateBuildInfo('addIncludePaths','$(START_DIR)/codegen/lib/myabsval');
        coder.cinclude('myabsval_initialize.h');
        coder.cinclude('myabsval.h');
        coder.cinclude('myabsval_terminate.h');

        % Executing in the generated code.
        % Call the initialize function before calling the
        % C function for the first time
        coder.ceval('myabsval_initialize');

        % Call the generated C library function `myabsval`
        y = coder.ceval('myabsval',y);

        % Call the terminate function after
        % calling the C function for the last time
        coder.ceval('myabsval_terminate');
    end

Generate the MEX function `callmyabsval_mex`. Provide the generated library file at the command line.
codegen -config:mex callmyabsval codegen\lib\myabsval\myabsval.lib -args \{-2.75\}

Rather than providing the library at the command line, you can use coder.updateBuildInfo to specify the library within the function. Use this option to preconfigure the build. Add this line to the else block:

coder.updateBuildInfo('addLinkObjects','myabsval.lib','$(START_DIR)\codegen\lib\myabsval',100,true,true);

Run the MEX function callmyabsval_mex which calls the library function myabsval.

callmyabsval_mex(-2.75)

ans =

   2.7500

Call the MATLAB function callmyabsval.

callmyabsval(-2.75)

ans =

   2.7500

The callmyabsval function exhibits the desired behavior for execution in MATLAB and in code generation.

Call C Function That Uses Global Variable

Use the '-global' flag when you call a C function that modifies a global variable.

Write a MATLAB function useGlobal that calls a C function addGlobal. Use the '-global' flag to indicate to the code generator that the C function uses a global variable.

function y = useGlobal()
    global g;
    t = g;
    % compare execution with/without '-global' flag
    coder.ceval('-global','addGlobal');
    y = t;
end

Create a C header file addGlobal.h for the function addGlobal.

2-75
void addGlobal(void);

Write the C function addGlobal in the file addGlobal.c. This function includes the header file useGlobal_data.h that the code generator creates when you generate code for the function useGlobal. This header file contains the global variable declaration for g.

#include "addGlobal.h"
#include "useGlobal_data.h"
void addGlobal(void) {
    g++;
}

Generate the MEX function for useGlobal. To define the input to the code generator, declare the global variable in the workspace.

global g;
g = 1;
codegen useGlobal -report addGlobal.h addGlobal.c
y = useGlobal_mex();

With the '-global' flag, the MEX function produces the result y = 1. The '-global' flag indicates to the code generator that the C function possibly modifies the global variable. For useGlobal, the code generator produces this code:

real_T useGlobal(const emlrtStack *sp)
{
    real_T y;
    (void)sp;
    y = g;
    addGlobal();
    return y;
}

Without the '-global' flag, the MEX function produces y = 2. Because there is no indication that the C function modifies g, the code generator assumes that y and g are identical. This C code is generated:

real_T useGlobal(const emlrtStack *sp)
{
    (void)sp;
    addGlobal();
}
Call C Function That Uses Different Array Layout

Suppose that you have a C function `testRM` that is designed to use row-major layout. You want to integrate this function into a MATLAB function `bar` that operates on arrays. The function `bar` is designed to use column-major layout, employing the `coder.columnMajor` directive.

```matlab
function out = bar(in)
%#codegen
coder.columnMajor;
coder.ceval('-layout:rowMajor','testRM', ...
    coder.rref(in),coder.wref(out));
end
```

In the generated code, the code generator inserts a layout conversion from column-major layout to row-major layout on the variable `in` before passing it to `testRM`. On the output variable `out`, the code generator inserts a layout conversion back to column-major.

In general, if you do not specify the `layout` option for `coder.ceval`, the external function arguments are assumed to use column-major.

Input Arguments

- **cfun_name — C/C++ function name**
  character vector | string scalar
  Name of external C/C++ function to call.
  Example: `coder.ceval('foo')`
  Data Types: char | string

- **cfun_arguments — C/C++ function arguments**
  scalar variable | array | element of an array | structure | structure field | object property
  Comma-separated list of input arguments in the order that `cfun_name` requires.
  Example: `coder.ceval('foo', 10, 20);`
Example: `coder.ceval('myFunction', coder.ref(x));`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `struct`

Complex Number Support: Yes

**Limitations**

- You cannot use `coder.ceval` on functions that you declare extrinsic with `coder.extrinsic`.
- When the LCC compiler creates a library, it adds a leading underscore to the library function names. If the compiler for the library was LCC and your code generation compiler is not LCC, you must add the leading underscore to the function name, for example, `coder.ceval('_mylibfun')`. If the compiler for a library was not LCC, you cannot use LCC to generate code from MATLAB code that calls functions from that library. Those library function names do not have the leading underscore that the LCC compiler requires.
- If a property has a get method, a set method, or validators, or is a System object property with certain attributes, then you cannot pass the property by reference to an external function. See “Passing By Reference Not Supported for Some Properties”.

**Tips**

- For code generation, before calling `coder.ceval`, you must specify the type, size, and complexity data type of return values and output arguments.
- To apply `coder.ceval` to a function that accepts or returns variables that do not exist in MATLAB code, such as pointers, FILE types for file I/O, and C/C++ macros, use the `coder.opaque` function.
- Use `coder.ceval` only in MATLAB for code generation. `coder.ceval` generates an error in uncompiled MATLAB code. To determine if a MATLAB function is executing in MATLAB, use `coder.target`. If the function is executing in MATLAB, call the MATLAB version of the C/C++ function.

**See Also**

`coder.ExternalDependency` | `coder.extrinsic` | `coder.opaque` | `coder.ref` | `coder.rref` | `coder.target` | `coder.updateBuildInfo` | `coder.wref`
Topics
“Integrate C Code Using the MATLAB Function Block”
“Generate Code That Uses Row-Major Array Layout” (MATLAB Coder)
“Interface with Row-Major Data in MATLAB Function Block”
“Unknown Output Type for coder.ceval”

Introduced in R2011a
**coder.cinclude**

Include header file in generated code

**Syntax**

```matlab
coder.cinclude(headerfile)
coder.cinclude(headerfile,'InAllSourceFiles',allfiles)
```

**Description**

coder.cinclude(headerfile) includes a header file in generated C/C++ source code.

MATLAB Coder generates the include statement in the C/C++ source files that are generated from the MATLAB code that contains the coder.cinclude call.

In a Simulink model, when a coder.cinclude call appears in a MATLAB Function block, the code generator puts the include statement in the model header file.

coder.cinclude(headerfile,'InAllSourceFiles',allfiles) uses the allfiles option to determine whether to include the header file in almost all C/C++ source files.

If allfiles is true, MATLAB Coder generates the include statement in almost all C/C++ source files, except for some utility files. This behavior is the coder.cinclude behavior from R2016a and earlier releases. The presence of the include statement in these additional files can increase compile time and make the generated code less readable. Use this option only if your code depends on the legacy behavior. If allfiles is false, the behavior is the same as the behavior of coder.cinclude(headerfile).

In a MATLAB Function block, coder.cinclude(headerfile,'InAllSourceFiles',allfiles) is the same as coder.cinclude(headerfile).

**Examples**
Include Header File in C/C++ Code Generated by Using the MATLAB Coder codegen Command

Generate code from a MATLAB function that calls an external C function. Use coder.cinclude to include the required header file in the generated C code.

In a writable folder, create a subfolder mycfiles.

Write a C function myMult2.c that doubles its input. Save it in mycfiles.

```c
#include "myMult2.h"
double myMult2(double u)
{
    return 2 * u;
}
```

Write the header file myMult2.h. Save it in mycfiles.

```c
#if !defined(MYMULT2)
#define MYMULT2
extern double myMult2(double);
#endif
```

Write a MATLAB function, myfunc, that includes myMult2.h and calls myMult2 for code generation only.

```matlab
function y = myfunc
%#codegen
y = 21;
if ~coder.target('MATLAB')
    % Running in generated code
    coder.cinclude('myMult2.h');
    y = coder.ceval('myMult2', y);
else
    % Running in MATLAB
    y = y * 2;
end
end
```

Create a code configuration object for a static library. Specify the locations of myMult2.h and myMult2.c

```matlab
cfg = coder.config('lib');
cfg.CustomInclude = fullfile(pwd,'mycfiles');
cfg.CustomSource = fullfile(pwd,'mycfiles','myMult2.c');
```
Generate the code.

codegen -config cfg myfunc -report

The file myfunc.c contains this statement:

#include "myMult2.h"

The include statement does not appear in any other file.

Include Header File in C/C++ Code Generated from a MATLAB Function Block in a Simulink Model

Generate code from a MATLAB Function block that calls an external C function. Use coder.cinclude to include the required header file in the generated C code.

In a writable folder, create a subfolder mycfiles.

Write a C function myMult2.c that doubles its input. Save it in mycfiles.

#include "myMult2.h"
double myMult2(double u)
{
    return 2 * u;
}

Write the header file myMult2.h. Save it in mycfiles.

#if !defined(MYMULT2)
#define MYMULT2
extern double myMult2(double);
#endif

Create a Simulink model that contains a MATLAB Function block connected to an Outport block.
In the MATLAB Function block, add the function `myfunc` that includes `myMult2.h` and calls `myMult2`.

```matlab
function y = myfunc
%#codegen
y = 21;
coder.cinclude('myMult2.h');
y = coder.ceval('myMult2', y);
% Specify the locations of myMult2.h and myMult2.c
coder.extrinsic('pwd', 'fullfile');
customDir = coder.const(fullfile(pwd, 'mycfiles'));
coder.updateBuildInfo('addIncludePaths', customDir);
coder.updateBuildInfo('addSourcePaths', customDir);
coder.updateBuildInfo('addSourceFiles', 'myMult2.c');
end
```

Open the Configuration Parameters dialog box.

On the **Solver** pane, select a fixed-step solver.

Save the model as `mymodel`.

Build the model.

The file `mymodel.h` contains this statement:

```
#include "myMult2.h"
```

To read more about integrating custom code in a MATLAB Function block, see “Integrate C Code Using the MATLAB Function Block”.

**Input Arguments**

- **headerfile** — Name of header file
  
  character vector | string scalar

  Name of a header file specified as a character vector or string scalar. `headerfile` must be a compile-time constant.

  Enclose a system header file name in angle brackets `< >`. The generated `#include` statement for a system header file has the format `#include <sysheader>`. A system header file must be in a standard location or on the include path. Specify the include path by using code generation custom code parameters.
Example: `coder.cinclude('<sysheader.h>')`

For a header file that is not a system header file, omit the angle brackets. The generated `#include` statement for a header file that is not a system header file has the format `#include "myHeader"`. The header file must be in the current folder or on the include path. Specify the include path by using code generation custom code parameters.

Example: `coder.cinclude('myheader.h')`

Data Types: `char`

**allfiles — All source files option**

true | false

Option to include header file in all generated C/C++ source files. If `allfiles` is `true`, MATLAB Coder generates the include statement in almost all of the C/C++ source files, except for some utility files. If `allfiles` is `false`, the behavior is the same as the behavior of `coder.cinclude(headerfile)`.

In a MATLAB Function block, the code generator ignores the all source files option.

Data Types: `logical`

**Limitations**

- Do not call `coder.cinclude` inside run-time conditional constructs such as `if` statements, `switch` statements, `while-loops`, and `for-loops`. You can call `coder.cinclude` inside compile-time conditional statements, such as `coder.target`. For example:

```...  
if ~coder.target('MATLAB')  
  coder.cinclude('foo.h');  
  coder.ceval('foo');  
end  
...```

**Tips**

- Before a `coder.ceval` call, call `coder.cinclude` to include the header file required by the external function that `coder.ceval` calls.
• Extraneous include statements in generated C/C++ code can increase compile time and reduce code readability. To avoid extraneous include statements in code generated by MATLAB Coder, follow these best practices:
  
  • Place a `coder.cinclude` call as close as possible to the `coder.ceval` call that requires the header file.
  • Do not set `allfiles` to `true`.

For the MATLAB Function block, the code generator generates the include statement in the model header file.

• In R2016a and earlier releases, for any `coder.cinclude` call, MATLAB Coder included the header file in almost all generated C/C++ source files, except for some utility files. If you have code that depends on this legacy behavior, you can preserve the legacy behavior by using this syntax:

```
coder.cinclude(headerfile,'InAllSourceFiles',true)
```

**See Also**
`coder.ceval` | `coder.target`

**Topics**
“Model Configuration Parameters: Code Generation Custom Code” (Simulink Coder)

**Introduced in R2013a**
**coder.columnMajor**

Specify column-major array layout for a function or class

**Syntax**

coder.columnMajor

**Description**

coder.columnMajor specifies column-major array layout for the data used by the current function in generated code. When placed in a class constructor, coder.columnMajor specifies column-major layout for data used by the class.

**Examples**

**Specify Column-Major Array Layout for a Function**

Specify column-major array layout for a function by inserting coder.columnMajor into the function body.

Suppose that myFunction is the top-level function of your code. Your application requires you to perform matrix addition with column-major array layout and matrix multiplication with row-major layout.

```matlab
function S = myFunction(A,B)
    %#codegen
    if size(A,1) ~= size(B,1) || size(A,2) ~= size(B,2)
        disp('Matrices must be same size.')
        return;
    end
    B = B*B';
    A = A*A';
```
% add matrices
S = addMatrix(A,B);
end

Write a function for matrix addition called addMatrix. Specify column-major for addMatrix by using coder.columnMajor.

function S = addMatrix(A,B)
%#codegen
S = zeros(size(A));
coder.columnMajor; % specify column-major array layout
S = A + B;
end

Generate code for myFunction. Use the codegen command.

codegen myFunction -args {ones(10,20),ones(10,20)} -config:lib -launchreport -rowmajor

Because of the codegen -rowmajor option, the matrix multiplication in myFunction uses row-major layout. However, the generated code for addMatrix uses column-major array layout due to the coder.columnMajor call.

Tips

• The code generator uses column-major array layout by default.
• The specification of array layout inside a function supersedes the array layout specified with the codegen command. For example, if the function foo contains coder.columnMajor, and you generate code by using:

codegen foo -rowmajor

then the generated code still uses column-major layout.
• Other functions called from within a column-major function inherit the column-major specification. However, if one of the called functions has its own distinct coder.rowMajor call, the code generator changes the array layout accordingly. If a row-major function and a column-major function call the same function, which does not have its own array layout specification, the code generator produces a row-major version and column-major version of the function.
• coder.columnMajor is ignored outside of code generation and simulation.
See Also

coder.ceval | coder.isColumnMajor | coder.isRowMajor | coder.rowMajor

Topics

“Interface with Row-Major Data in MATLAB Function Block”
“Specify Array Layout in Functions and Classes”
“Code Generation of Matrices and Arrays” (Simulink Coder)

Introduced in R2018a
**coder.const**

Fold expressions into constants in generated code

**Syntax**

\[ \text{out} = \text{coder.const(} \text{expression} \text{)} \]

\[ [\text{out1},...,\text{outN}] = \text{coder.const(} \text{handle,} \text{arg1},...,\text{argN} \text{)} \]

**Description**

\( \text{out} = \text{coder.const(} \text{expression} \text{)} \) evaluates \( \text{expression} \) and replaces \( \text{out} \) with the result of the evaluation in generated code.

\[ [\text{out1},...,\text{outN}] = \text{coder.const(} \text{handle,} \text{arg1},...,\text{argN} \text{)} \] evaluates the multi-output function having handle \( \text{handle} \). It then replaces \( \text{out1},...,\text{outN} \) with the results of the evaluation in the generated code.

**Examples**

**Specify Constants in Generated Code**

This example shows how to specify constants in generated code using \texttt{coder.const}.

Write a function AddShift that takes an input \texttt{Shift} and adds it to the elements of a vector. The vector consists of the square of the first 10 natural numbers. AddShift generates this vector.

\begin{verbatim}
function y = AddShift(Shift) %#codegen
y = (1:10).^2+Shift;
\end{verbatim}

Generate code for AddShift using the \texttt{codegen} command. Open the Code Generation Report.

\begin{verbatim}
codegen -config:lib -launchreport AddShift -args 0
\end{verbatim}
The code generator produces code for creating the vector. It adds Shift to each element of the vector during vector creation. The definition of AddShift in generated code looks as follows:

```c
void AddShift(double Shift, double y[10])
{
    int k;
    for (k = 0; k < 10; k++) {
        y[k] = (double)((1 + k) * (1 + k)) + Shift;
    }
}
```

Replace the expression `(1:10).^2` with `coder.const((1:10).^2)`, and then generate code for AddShift again using the codegen command. Open the Code Generation Report.

```bash
codegen -config:lib -launchreport AddShift -args 0
```

The code generator creates the vector containing the squares of the first 10 natural numbers. In the generated code, it adds Shift to each element of this vector. The definition of AddShift in generated code looks as follows:

```c
void AddShift(double Shift, double y[10])
{
    int i;
    static const signed char iv[10] = { 1, 4, 9, 16, 25, 36,
    49, 64, 81, 100 };

    for (i = 0; i < 10; i++) {
        y[i] = (double)iv[i] + Shift;
    }
}
```

Create Lookup Table in Generated Code

This example shows how to fold a user-written function into a constant in generated code.

Write a function `getsine` that takes an input `index` and returns the element referred to by `index` from a lookup table of sines. The function `getsine` creates the lookup table using another function `gettable`.

```matlab
function y = getsine(index) %#codegen
    assert(isa(index, 'int32'))
```
persistent tbl;
if isempty(tbl)
    tbl = gettable(1024);
end
y = tbl(index);

function y = gettable(n)
    y = zeros(1,n);
    for i = 1:n
        y(i) = sin((i-1)/(2*pi*n));
    end
end

Generate code for getsine using an argument of type int32. Open the Code Generation Report.

codegen -config:lib -launchreport getsine -args int32(0)

The generated code contains instructions for creating the lookup table.

Replace the statement:

tbl = gettable(1024);

with:

tbl = coder.const(gettable(1024));

Generate code for getsine using an argument of type int32. Open the Code Generation Report.

The generated code contains the lookup table itself. coder.const forces the expression gettable(1024) to be evaluated during code generation. The generated code does not contain instructions for the evaluation. The generated code contains the result of the evaluation itself.

**Specify Constants in Generated Code Using Multi-Output Function**

This example shows how to specify constants in generated code using a multi-output function in a coder.const statement.
Write a function `MultiplyConst` that takes an input `factor` and multiplies every element of two vectors `vec1` and `vec2` with `factor`. The function generates `vec1` and `vec2` using another function `EvalConsts`.

```matlab
function [y1,y2] = MultiplyConst(factor) %#codegen
    [vec1,vec2]=EvalConsts(pi.*(1./2.^(1:10)),2);
    y1=vec1.*factor;
    y2=vec2.*factor;
end

function [f1,f2]=EvalConsts(z,n)
    f1=z.^(2*n)/factorial(2*n);
    f2=z.^(2*n+1)/factorial(2*n+1);
end
```

Generate code for `MultiplyConst` using the `codegen` command. Open the Code Generation Report.

codegen -config:lib -launchreport MultiplyConst -args 0

The code generator produces code for creating the vectors. Replace the statement

```
[vec1,vec2]=EvalConsts(pi.*(1./2.^(1:10)),2);
```

with

```
[vec1,vec2]=coder.const(@EvalConsts,pi.*(1./2.^(1:10)),2);
```

Generate code for `MultiplyConst` using the `codegen` command. Open the Code Generation Report.

codegen -config:lib -launchreport MultiplyConst -args 0

The code generator does not generate code for creating the vectors. Instead, it calculates the vectors and specifies the calculated vectors in generated code.

**Read Constants by Processing XML File**

This example shows how to call an extrinsic function using `coder.const`.

Write an XML file `Myparsr.xml` containing the following statements:

```
<params>
    <param name="hello" value="17"/>
</params>
```
Save MyParams.xml in the current folder.

Write a MATLAB function `xml2struct` that reads an XML file. The function identifies the XML tag `param` inside another tag `params`.

After identifying `param`, the function assigns the value of its attribute `name` to the field `name` of a structure `s`. The function also assigns the value of attribute `value` to the value of the field.

```matlab
function s = xml2struct(file)
    s = struct();
    doc = xmlread(file);
    els = doc.getElementsByTagName('params');
    for i = 0:els.getLength-1
        it = els.item(i);
        ps = it.getElementsByTagName('param');
        for j = 0:ps.getLength-1
            param = ps.item(j);
            paramName = char(param.getAttribute('name'));
            paramValue = char(param.getAttribute('value'));
            paramValue = evalin('base', paramValue);
            s.(paramName) = paramValue;
        end
    end
end
```

Save `xml2struct` in the current folder.

Write a MATLAB function `MyFunc` that reads the XML file `MyParams.xml` into a structure `s` using the function `xml2struct`. Declare `xml2struct` as extrinsic using `coder.extrinsic` and call it in a `coder.const` statement.

```matlab
function y = MyFunc(u) %#codegen
    assert(isa(u, 'double'));
    coder.extrinsic('xml2struct');
    s = coder.const(xml2struct('MyParams.xml'));
    y = s.hello + s.world + u;
end
```

Generate code for `MyFunc` using the `codegen` command. Open the Code Generation Report.

codegen -config:dll -launchreport MyFunc -args 0
The code generator executes the call to `xml2struct` during code generation. It replaces the structure fields `s.hello` and `s.world` with the values 17 and 42 in generated code.

**Input Arguments**

*expression* — MATLAB expression or user-written function

expression with constants | single-output function with constant arguments

MATLAB expression or user-defined single-output function.

The expression must have compile-time constants only. The function must take constant arguments only. For instance, the following code leads to a code generation error, because `x` is not a compile-time constant.

```matlab
function y=func(x)
    y=coder.const(log10(x));
```

To fix the error, assign `x` to a constant in the MATLAB code. Alternatively, during code generation, you can use `coder.Constant` to define input type as follows:

```bash
codegen -config:lib func -args coder.Constant(10)
```

Example: `2*pi`, `factorial(10)`

**handle** — Function handle

Handle to built-in or user-written function.

Example: `@log`, `@sin`

Data Types: `function_handle`

*arg1,...,argN* — Arguments to the function with handle handle

Arguments to the function with handle `handle`.

The arguments must be compile-time constants. For instance, the following code leads to a code generation error, because `x` and `y` are not compile-time constants.

```matlab
function y=func(x,y)
    y=coder.const(@nchoosek,x,y);
```
To fix the error, assign \( x \) and \( y \) to constants in the MATLAB code. Alternatively, during code generation, you can use `coder.Constant` to define input type as follows:

```
codegen -config:lib func -args {coder.Constant(10),coder.Constant(2)}
```

## Output Arguments

- **out** — Value of expression
  value of the evaluated expression

Value of expression. In the generated code, MATLAB Coder replaces occurrences of `out` with the value of expression.

- **out1,...,outN** — Outputs of the function with handle handle
  values of the outputs of the function with handle handle

Outputs of the function with handle handle. MATLAB Coder evaluates the function and replaces occurrences of `out1,...,outN` with constants in the generated code.

## Tips

- When possible, the code generator constant-folds expressions automatically. Typically, automatic constant-folding occurs for expressions with scalars only. Use `coder.const` when the code generator does not constant-fold expressions on its own.

- When constant-folding computationally intensive function calls, to reduce code generation time, make the function call extrinsic. The extrinsic function call causes evaluation of the function call by MATLAB instead of by the code generator. For example:

  ```
  function j = fcn(z)
  zTable = coder.const(0:0.01:100);
  jTable = coder.const(feval('besselj',3,zTable));
  j = interp1(zTable,jTable,z);
  end
  ```

  See “Use coder.const with Extrinsic Function Calls” (MATLAB Coder).

- If `coder.const` is unable to constant-fold a function call, try to force constant-folding by making the function call extrinsic. The extrinsic function call causes evaluation of the function call by MATLAB instead of by the code generator. For example:
function yi = fcn(xi)
y = coder.const(feval('rand',1,100));
yi = interp1(y,xi);
end

See “Use coder.const with Extrinsic Function Calls” (MATLAB Coder).

See Also

Topics
“Fold Function Calls into Constants” (MATLAB Coder)
“Use coder.const with Extrinsic Function Calls” (MATLAB Coder)

Introduced in R2013b
**coder.cstructname**

Package: coder

Name C structure type in generated code

**Syntax**

```matlab
coder.cstructname(var, structName)
coder.cstructname(var, structName, 'extern', 'HeaderFile', headerfile)
coder.cstructname(var, structName, 'extern', 'HeaderFile',
                 headerfile, 'Alignment', alignment)
```

```matlab
outtype = coder.cstructname(intype, structName)
outtype = coder.cstructname(intype, structName, 'extern', 'HeaderFile',
                             headerfile)
outtype = coder.cstructname(intype, structName, 'extern', 'HeaderFile',
                             headerfile, 'Alignment', alignment)
```

**Description**

coder.cstructname names the generated or externally defined C structure type to use for MATLAB variables that are represented as structures in generated code.

coder.cstructname(var, structName) names the C structure type generated for the MATLAB variable var. The input var can be a structure or a cell array. Use this syntax in a function from which you generate code. Place coder.cstructname after the definition of var and before the first use of var. If var is an entry-point (top-level) function input argument, place coder.cstructname at the beginning of the function, before any control flow statements.

coder.cstructname(var, structName, 'extern', 'HeaderFile', headerfile) specifies that the C structure type to use for var has the name structName and is defined in the external file, headerfile.
It is possible to use the 'extern' option without specifying the header file. However, it is a best practice to specify the header file so that the code generator produces the `#include` statement in the correct location.

coder.cstructname(var,structName,'extern','HeaderFile', headerfile,'Alignment',alignment) also specifies the run-time memory alignment for the externally defined structure type structName. If you have Embedded Coder and use custom Code Replacement Libraries (CRLs), specify the alignment so that the code generator can match CRL functions that require alignment for structures. See “Data Alignment for Code Replacement” (Embedded Coder).

outtype = coder.cstructname(intype,structName) returns a structure or cell array type object outtype that specifies the name of the C structure type to generate. coder.cstructname creates outtype with the properties of the input type intype. Then, it sets the TypeName property to structName. Use this syntax to create a type object that you use with the codegen -args option. You cannot use this syntax in a function from which you generate code.

You cannot use this syntax in a MATLAB Function block.

outtype = coder.cstructname(intype,structName,'extern','HeaderFile', headerfile) returns a type object outtype that specifies the name and location of an externally defined C structure type. The code generator uses the externally defined structure type for variables with type outtype.

You cannot use this syntax in a MATLAB Function block.

outtype = coder.cstructname(intype,structName,'extern','HeaderFile', headerfile,'Alignment',alignment) creates a type object outtype that also specifies the C structure type alignment.

You cannot use this syntax in a MATLAB Function block.

**Examples**

**Name the C Structure Type for a Variable in a Function**

In a MATLAB function, `myfun`, assign the name `MyStruct` to the generated C structure type for the variable `v`. 
function y = myfun()
    %#codegen
    v = struct('a',1,'b',2);
    coder.cstructname(v, 'myStruct');
    y = v;
end

Generate standalone C code. For example, generate a static library.

codegen -config:lib myfun -report

To see the generated structure type, open codegen/lib/myfun/myfun_types.h or view myfun_types.h in the code generation report. The generated C structure type is:

typedef struct {
    double a;
    double b;
} myStruct;

**Name the C Structure Type Generated for a Substructure**

In a MATLAB function, myfun1, assign the name MyStruct to the generated C structure type for the structure v. Assign the name mysubStruct to the structure type generated for the substructure v.b.

function y = myfun()
    %#codegen
    v = struct('a',1,'b',struct('f',3));
    coder.cstructname(v, 'myStruct');
    coder.cstructname(v.b, 'mysubStruct');
    y = v;
end

The generated C structure type mysubStruct is:

typedef struct {
    double f;
} mysubStruct;

The generated C structure type myStruct is:

typedef struct {
    double a;
} myStruct;
mysubStruct b;
} myStruct;

Name the Structure Type Generated for a Cell Array

In a MATLAB function, myfun2, assign the name myStruct to the generated C structure type for the cell array c.

```matlab
function z = myfun2()
c = {1 2 3};
coder.cstructname(c,'myStruct')
z = c;
```

The generated C structure type for c is:

```c
typedef struct {
   double f1;
   double f2;
   double f3;
} myStruct;
```

Name an Externally Defined C Structure Type

Specify that a structure passed to a C function has a structure type defined in a C header file.

Create a C header file mycadd.h for the function mycadd that takes a parameter of type mycstruct. Define the type mycstruct in the header file.

```c
#ifndef MYCADD_H
#define MYCADD_H

typedef struct {
   double f1;
   double f2;
} mycstruct;

double mycadd(mycstruct *s);
#endif
```

Write the C function mycadd.c.
#include <stdio.h>
#include <stdlib.h>
#include "mycadd.h"

double mycadd(mycstruct *s)
{
    return s->f1 + s->f2;
}

Write a MATLAB function mymAdd that passes a structure by reference to mycadd. Use coder.cstructname to specify that in the generated code, the structure has the C type mycstruct, which is defined in mycadd.h.

function y = mymAdd
%#codegen
s = struct('f1', 1, 'f2', 2);
coder.cstructname(s, 'mycstruct', 'extern', 'HeaderFile', 'mycadd.h');
y = 0;
y = coder.ceval('mycadd', coder.ref(s));

Generate a C static library for function mymAdd.

codegen -config:lib mymAdd mycadd.c

The generated header file mymadd_types.h does not contain a definition of the structure mycstruct because mycstruct is an external type.

**Create a Structure Type Object That Names the Generated C Structure Type**

Suppose that the entry-point function myFunction takes a structure argument. To specify the type of the input argument at the command line:

1. Define an example structure S.
2. Create a type T from S by using coder.typeof.
3. Use coder.cstructname to create a type T1 that:
   - Has the properties of T.
   - Names the generated C structure type myStruct.
4. Pass the type to codegen by using the -args option.
For example:

```matlab
S = struct('a',double(0),'b',single(0));
T = coder.typeof(S);
T1 = coder.cstructname(T,'myStruct');
codegen -config:lib myFunction -args T1
```

Alternatively, you can create the structure type directly from the example structure.

```matlab
S = struct('a',double(0),'b',single(0));
T1 = coder.cstructname(S,'myStruct');
codegen -config:lib myFunction -args T1
```

### Input Arguments

- **var** — MATLAB structure or cell array variable
  
  structure | cell array

  MATLAB structure or cell array variable that is represented as a structure in the generated code.

- **structName** — Name of C structure type
  
  character vector | string scalar

  Name of generated or externally defined C structure type, specified as a character vector or string scalar.

- **headerfile** — Header file that contains the C structure type definition
  
  character vector | string scalar

  Header file that contains the C structure type definition, specified as a character vector or string scalar.

To specify the path to the file:

- Use the `codegen -I` option or the **Additional include directories** parameter on the MATLAB Coder app settings **Custom Code** tab.
- For a MATLAB Function block, on the **Simulation Target** and the **Code Generation** > **Custom Code** panes, under **Additional build information**, set the **Include directories** parameter.

Alternatively, use `coder.updateBuildInfo` with the 'addIncludePaths' option.
Example: 'mystruct.h'

**alignment** — Run-time memory alignment for structure
-1 (default) | power of 2 not greater than 128

Run-time memory alignment for generated or externally defined structure.

**intype** — Type object or variable for creation of new type object
coder StructType | coder CellType | structure | cell array

Structure type object, cell array type object, structure variable, or cell array variable from which to create a type object.

**Limitations**

- You cannot apply `coder.cstructname` directly to a global variable. To name the structure type to use with a global variable, use `coder.cstructname` to create a type object that names the structure type. Then, when you run `codegen`, specify that the global variable has that type. See “Name the C Structure Type to Use With a Global Structure Variable” (MATLAB Coder).
- For cell array inputs, the field names of externally defined structures must be `f1`, `f2`, and so on.

**Tips**

- For information about how the code generator determines the C/C++ types of structure fields, see “Mapping MATLAB Types to Types in Generated Code” (MATLAB Coder).
- Using `coder.cstructname` on a structure array sets the name of the structure type of the base element, not the name of the array. Therefore, you cannot apply `coder.cstructname` to a structure array element, and then apply it to the array with a different C structure type name. For example, the following code is not allowed. The second `coder.cstructname` attempts to set the name of the base type to `myStructArrayName`, which conflicts with the previously specified name, `myStructName`.

```matlab
% Define scalar structure with field a
myStruct = struct('a', 0);
```
coder.cstructname(myStruct,'myStructName');
% Define array of structure with field a
myStructArray = repmat(myStruct,4,6);
coder.cstructname(myStructArray,'myStructArrayName');

- Applying `coder.cstructname` to an element of a structure array produces the same result as applying `coder.cstructname` to the entire structure array. If you apply `coder.cstructname` to an element of a structure array, you must refer to the element by using a single subscript. For example, you can use `var(1)`, but not `var(1,1)`. Applying `coder.cstructname` to `var(:)` produces the same result as applying `coder.cstructname` to `var` or `var(n)`. 

- Heterogeneous cell arrays are represented as structures in the generated code. Here are considerations for using `coder.cstructname` with cell arrays:
  - In a function from which you generate code, using `coder.cstructname` with a cell array variable makes the cell array heterogeneous. Therefore, if a cell array is an entry-point function input and its type is permanently homogeneous, then you cannot use `coder.cstructname` with the cell array.
  - Using `coder.cstructname` with a homogeneous `coder.CellType` object `intype` makes the returned object heterogeneous. Therefore, you cannot use `coder.cstructname` with a permanently homogeneous `coder.CellType` object. For information about when a cell array is permanently homogeneous, see “Specify Cell Array Inputs at the Command Line” (MATLAB Coder).
  - When used with a `coder.CellType` object, `coder.cstructname` creates a `coder.CellType` object that is permanently heterogeneous.
  - When you use a structure named by `coder.cstructname` in a project with row-major and column-major array layouts, the code generator renames the structure in certain cases, appending `row_` or `col_` to the beginning of the structure name. This renaming provides unique type definitions for the types that are used in both array layouts.
  - These tips apply only to MATLAB Function blocks:
    - MATLAB Function block input and output structures are associated with bus signals. The generated name for the structure type comes from the bus signal name. Do not use `coder.cstructname` to name the structure type for input or output signals. See “Create Structures in MATLAB Function Blocks”.
    - The code generator produces structure type names according to identifier naming rules, even if you name the structure type with `coder.cstructname`. If you have Embedded Coder, you can customize the naming rules. See “Construction of Generated Identifiers” (Embedded Coder).
See Also
coder.ceval

Topics
“Structure Definition for Code Generation”
“Code Generation for Cell Arrays”
“Integrate C Code Using the MATLAB Function Block”

Introduced in R2011a
**coder.extrinsic**

Declare extrinsic functions

**Syntax**

```matlab
coder.extrinsic(function)
coder.extrinsic(function1, ... ,functionN)
coder('-sync:on', function1, ... ,functionN)
coder('-sync:off', function1, ... ,functionN)
```

**Description**

`coder.extrinsic(function)` declares `function` as an extrinsic function. The code generator does not produce code for the body of the extrinsic function and instead uses the MATLAB engine to execute the call. This functionality is available only when the MATLAB engine is available in MEX functions, Simulink simulations, or during calls at compile time.

During standalone code generation, the code generator attempts to determine whether an extrinsic function affects the output of the function in which it is called — for example by returning `mxArrays` to an output variable. Provided that there is no change to the output, the code generator proceeds with code generation, but excludes the extrinsic function from the generated code. Otherwise, the code generator produces a compilation error.

You cannot use `coder.ceval` on functions that you declare as extrinsic by using `coder.extrinsic`.

`coder.extrinsic` is ignored outside of code generation.

`coder.extrinsic(function1, ... ,functionN)` declares `function1` through `functionN` as extrinsic functions.

`coder('-sync:on', function1, ... ,functionN)` enables synchronization of global data between MATLAB and MEX functions before and after calls to the extrinsic functions `function1` through `functionN`. If only a few extrinsic calls use or modify
global data, turn off synchronization before and after all extrinsic function calls by setting the global synchronization mode to `At MEX-function entry and exit`. Use the `'-sync:on'` option to turn on synchronization for only the extrinsic calls that do modify global data.

See “Generate Code for Global Data” (MATLAB Coder).

```matlab
coder('-sync:off', function1, ... ,functionN)
```
disables synchronization of global data between MATLAB and MEX functions before and after calls to the extrinsic functions `function1` through `functionN`. If most extrinsic calls use or modify global data, but a few do not, use the `'-sync:off'` option to turn off synchronization for the extrinsic calls that do not modify global data.

See “Generate Code for Global Data” (MATLAB Coder).

## Examples

### Declare a MATLAB Function as Extrinsic

The MATLAB function `patch` is not supported for code generation. This example shows how you can still use the functionality of `patch` in your generated MEX function by declaring `patch` as extrinsic your MATLAB function.

This MATLAB code declares `patch` as extrinsic in the local function `create_plot`. By declaring `patch` as extrinsic, you instruct the code generator not to produce code for `patch`. Instead, the code generator dispatches `patch` to MATLAB for execution.

The code generator automatically treats many common MATLAB visualization functions, such as the function `axis` as extrinsic.

```matlab
function c = pythagoras(a,b,color) %#codegen
% Calculate the hypotenuse of a right triangle
% and display the triangle as a patch object.
c = sqrt(a^2 + b^2);
create_plot(a, b, color);
end

function create_plot(a, b, color)
%Declare patch as extrinsic
coder.extrinsic('patch');
```
\begin{verbatim}
x = [0;a;a];
y = [0;0;b];
patch(x,y,color);
axis('equal');
end
\end{verbatim}

Generate a MEX function for \textit{pythagoras}. Also, generate the code generation report.

codegen -report pythagoras -args {1, 1, [.3 .3 .3]}

In the report, view the MATLAB code for \textit{create_plot}.

\begin{verbatim}
function create_plot(a, b, color)
coder.extrinsic('patch');
x = [0;a;a];
y = [0;0;b];
patch(x,y,color);
axis('equal');
end
\end{verbatim}

The report highlights the \textit{patch} and \textit{axis} functions to indicate that they are treated as extrinsic functions.

Run the MEX function.

\texttt{pythagoras_mex(3, 4, [1.0 0.0 0.0]);}

MATLAB displays the plot of the right triangle as a red patch object.
Input Arguments

function — MATLAB function name
character vector

Name of the MATLAB function that is declared as extrinsic.

Example: coder.extrinsic('patch')

Data Types: char
Limitations

- Extrinsic function calls have some overhead that can affect performance. Input data that is passed in an extrinsic function call must be provided to MATLAB, which requires making a copy of the data. If the function has any output data, this data must be transferred back into the MEX function environment, which also requires a copy.
- The code generator does not support the use of `coder.extrinsic` to call functions that are located in a private folder.
- The code generator does not support the use of `coder.extrinsic` to call local functions.

Tips

- The code generator automatically treats many common MATLAB visualization functions, such as `plot`, `disp`, and `figure`, as extrinsic. You do not have to explicitly declare them as extrinsic functions by using `coder.extrinsic`.
- Use the `coder.screener` function to detect which functions you must declare as extrinsic. This function runs the Code Generation Readiness Tool that screens the MATLAB code for features and functions that are not supported for code generation.

See Also

coder.ceval | coder.screener

Topics

“Extrinsic Functions”
“Generate Code for Global Data” (MATLAB Coder)
“Resolution of Function Calls for Code Generation”

Introduced in R2011a
coder.ignoreConst

Prevent use of constant value of expression for function specializations

Syntax

coder.ignoreConst(expression)

Description

coder.ignoreConst(expression) prevents the code generator from using the constant value of expression to create function specializations on page 2-113. coder.ignoreConst(expression) returns the value of expression.

Examples

Prevent Function Specializations Based on Constant Input Values

Use coder.ignoreConst to prevent function specializations for a function that is called with constant values.

Write the function call_myfn, which calls myfcn.

```matlab
function [x, y] = call_myfcn(n)
%#codegen
x = myfcn(n, 'mode1');
y = myfcn(n, 'mode2');
end

function y = myfcn(n,mode)
coder.inline('never');
if strcmp(mode,'mode1')
    y = n;
else
    y = -n;
end
```
Generate standalone C code. For example, generate a static library. Enable the code generation report.

codegen -config:lib call_myfcn -args {1} -report

In the code generation report, you see two function specializations for call_myfcn.

The code generator creates call_myfcn>myfcn>1 for mode with a value of 'mode1'. It creates call_myfcn>myfcn>2 for mode with a value of 'mode2'.

In the generated C code, you see the specializations my_fcn and b_my_fcn.

static double b_myfcn(double n)
{
    return -n;
}

static double myfcn(double n)
{
    return n;
}

To prevent the function specializations, instruct the code generator to ignore that values of the mode argument are constant.

function [x, y] = call_myfcn(n)
%#codegen
x = myfcn(n, coder.ignoreConst('mode1'));
y = myfcn(n, coder.ignoreConst('mode2'));
end

function y = myfcn(n,mode)
coder.inline('never');
if strcmp(mode, 'mode1')
y = n;
else
  y = -n;
end
end

Generate the C code.

codegen -config:lib call_myfcn -args {1} -report

In the code generation report, you do not see multiple function specializations.

In the generated C code, you see one function for my_fcn.

**Input Arguments**

expression — Expression whose value is to be treated as a nonconstant

MATLAB expression

**More About**

**Function Specialization**

Version of a function in which an input type, size, complexity, or value is customized for a particular invocation of the function.

Function specialization produces efficient C code at the expense of code duplication. The code generation report shows all MATLAB function specializations that the code generator creates. However, the specializations might not appear in the generated C/C++ code due to later transformations or optimizations.
**Tips**

- For some recursive function calls, you can use `coder.ignoreConst` to force run-time recursion. See “Force Code Generator to Use Run-Time Recursion”.
- `coder.ignoreConst(expression)` prevents the code generator from using the constant value of `expression` to create function specializations. It does not prevent other uses of the constant value during code generation.

**See Also**
coder.ignoreSize | coder.inline

**Topics**
“Force Code Generator to Use Run-Time Recursion”
“Compile-Time Recursion Limit Reached”
“Avoid Duplicate Functions in Generated Code”

**Introduced in R2017a**
**coder.ignoreSize**

**Package:** coder

Prevent code generator from creating function specializations for constant-size expressions

**Syntax**

coder.ignoreSize(expression)

**Description**

coder.ignoreSize(expression) declares that the code generator must not use the constant size of an expression to create function specializations.

**Examples**

**Duplicate Functions Generated for Multiple Input Sizes**

If your MATLAB code calls a function multiple times and passes inputs of different sizes, the code generator can create function specializations for each size. To avoid this issue, use coder.ignoreSize on the function input. For example, this code uses coder.ignoreSize to avoid creating multiple copies of the function indexOf:

```matlab
function [out1, out2] = test1(in)
    a = 1:10;
    b = 2:40;
    % Without coder.ignoreSize duplicate functions are generated
    out1 = indexOf(coder.ignoreSize(a), in);
    out2 = indexOf(coder.ignoreSize(b), in);
end
```

```matlab
function index = indexOf(array, value)
    coder.inline('never');
    for i = 1:numel(array)
    end
```
if array(i) == value
    index = i;
    return
end
end
index = -1;
return
end

To generate code, enter:

codegen test1 -config:lib -report -args {1}

**Input Arguments**

expression — Expression whose size is to be treated as nonconstant MATLAB expression

Example: foo(coder.ignoreSize(1:10))

**More About**

**Function Specialization**

Version of a function in which an input type, size, complexity, or value is customized for a particular invocation of the function.

Function specialization produces efficient C code at the expense of code duplication. The code generation report shows all MATLAB function specializations that the code generator creates. However, the specializations might not appear in the generated C/C++ code due to later transformations or optimizations.

**Tips**

- If you assign an expression to a variable and declare the variable as variable-size by using coder.varsize, this declaration has the same effect as using coder.ignoreSize on the expression.
See Also
coder.ignoreConst

Topics
“Avoid Duplicate Functions in Generated Code”

Introduced in R2019b
**coder.inline**

**Package:** coder

Control inlining in generated code

**Syntax**

coder.inline('always')
coder.inline('never')
coder.inline('default')

**Description**

coder.inline('always') forces inlining on page 2-119 of the current function in the generated code. Place the `coder.inline` directive inside the function to which it applies. The code generator does not inline entry-point functions, inline functions into `parfor` loops, or inline functions called from `parfor` loops.

coder.inline('never') prevents inlining of the current function in the generated code. Prevent inlining when you want to simplify the mapping between the MATLAB source code and the generated code.

coder.inline('default') uses internal heuristics to determine whether to inline the current function. Usually, the heuristics produce highly optimized code. Use `coder.inline` only when you need to fine-tune these optimizations.

**Examples**

- “Prevent Function Inlining” on page 2-118
- “Use coder.inline in Control Flow Statements” on page 2-119

**Prevent Function Inlining**

In this example, function `foo` is not inlined in the generated code:
function y = foo(x)
    coder.inline('never');
    y = x;
end

Use coder.inline in Control Flow Statements

You can use coder.inline in control flow code. If the software detects contradictory coder.inline directives, the generated code uses the default inlining heuristic and issues a warning.

Suppose that you want to generate code for a division function used by a system with limited memory. To optimize memory use in the generated code, the inline_division function manually controls inlining based on whether it performs scalar division or vector division:

function y = inline_division(dividend, divisor)

% For scalar division, inlining produces smaller code
% than the function call itself.
if iscalar(dividend) && iscalar(divisor)
    coder.inline('always');
else
    % Vector division produces a for-loop.
    % Prohibit inlining to reduce code size.
    coder.inline('never');
end

if any(divisor == 0)
    error('Cannot divide by 0');
end

y = dividend / divisor;

More About

Inlining

Technique that replaces a function call with the contents (body) of that function. Inlining eliminates the overhead of a function call, but can produce larger C/C++ code. Inlining can create opportunities for further optimization of the generated C/C++ code.
See Also

Introduced in R2011a
coder.isColumnMajor

Determine whether the current function or variable uses column-major layout

Syntax

coder.isColumnMajor
coder.isColumnMajor(arg)

Description

coder.isColumnMajor resolves as true in the generated code if the current function uses column-major array layout. Use the function as the expression in control flow (if, else, switch) statements.

coder.isColumnMajor(arg) resolves as true if the current variable uses column-major array layout.

Examples

Query Array Layout of a Function

To query the array layout of a function at compile time, use coder.isColumnMajor or coder.isRowMajor. This query can be useful for specializing your generated code when it involves row-major and column-major functions. For example, consider this function:

```
function S = addMatrixRouted(A,B)
  if coder.isRowMajor
    %execute this code if row major
    S = addMatrix_OptimizedForRowMajor(A,B);
  elseif coder.isColumnMajor
    %execute this code if column major
    S = addMatrix_OptimizedForColumnMajor(A,B);
  end
```
The function `addMatrixRouted` behaves differently depending on whether it uses row-major layout or column-major layout. The layout that the function uses, for example, can depend on whether it is called from a function that contains `coder.rowMajor` or `coder.columnMajor`. When `addMatrixRouted` uses row-major layout, it calls the `addMatrix_OptimizedForRowMajor` function, which has efficient memory access for row-major data. When the function uses column-major layout, it calls a version of the `addMatrix` function optimized for column-major data.

By using the query functions, the generated code for `addMatrixRouted` provides efficient memory access for either choice of array layout.

**Query Array Layout of a Variable**

Consider the function `bar`:

```matlab
function bar
coder.columnMajor;
x = magic(3);
if coder.isColumnMajor(x)
    fprintf('This will always be displayed in generated code.
    
else
    fprintf('This will never be displayed in generated code.
    
end
```

Generate code:

```bash
codegen bar
```

To run the MEX function, enter:

```bash
bar_mex
```

**Input Arguments**

`arg` — Variable name
array variable

Variable to query for array layout.

Example: `coder.isColumnMajor(x);`
Limitations

- You cannot query the array layout of a structure field or property.

Tips

- The code generator uses column-major layout by default.
- Outside of code generation and simulation, `coder.isColumnMajor` is always true.
- If `coder.isColumnMajor` always resolves to true for your code, other branches in the if statement are ignored by the code generator. Otherwise, one instance of the current function is created for each array layout.

See Also

coder.ceval | coder.columnMajor | coder.isRowMajor | coder.rowMajor

Topics

“Interface with Row-Major Data in MATLAB Function Block”
“Specify Array Layout in Functions and Classes”
“Code Generation of Matrices and Arrays” (Simulink Coder)

Introduced in R2018a
**coder.isRowMajor**

Determine whether the current function or variable uses row-major layout

**Syntax**

coder.isRowMajor
coder.isRowMajor(arg)

**Description**

coder.isRowMajor resolves as true in the generated code if the current function uses row-major array layout. Use the function as the expression in control flow (if, else, switch) statements.

coder.isRowMajor(arg) resolves as true if the current variable uses row-major array layout.

**Examples**

**Query Array Layout of a Function**

To query the array layout of a function at compile time, use `coder.isRowMajor` or `coder.isColumnMajor`. This query can be useful for specializing your generated code when it involves row-major and column-major functions. For example, consider this function:

```matlab
function S = addMatrixRouted(A,B)
    if coder.isRowMajor
        %execute this code if row major
        S = addMatrix_OptimizedForRowMajor(A,B);
    elseif coder.isColumnMajor
        %execute this code if column major
        S = addMatrix_OptimizedForColumnMajor(A,B);
    end
```
The function `addMatrixRouted` behaves differently depending on whether it uses row-major layout or column-major layout. The layout that the function uses, for example, can depend on whether it is called from a function that contains `coder.rowMajor` or `coder.columnMajor`. When `addMatrixRouted` uses row-major layout, it calls the `addMatrix_OptimizedForRowMajor` function, which has efficient memory access for row-major data. When the function uses column-major layout, it calls a version of the `addMatrix` function optimized for column-major data.

By using the query functions, the generated code for `addMatrixRouted` provides efficient memory access for either choice of array layout.

**Query Array Layout of a Variable**

Consider the function `foo`:

```matlab
function foo
coder.rowMajor;
x = magic(3);
if coder.isRowMajor(x)
    fprintf('This will always be displayed in generated code.
');
else
    fprintf('This will never be displayed in generated code.
');
end
end
```

Generate code:

```
codegen foo
```

To run the MEX function, enter:

```
foo_mex
```

**Input Arguments**

`arg` — Variable name

Array variable

Variable to query for array layout.

Example: `coder.isRowMajor(x);`
Limitations

- You cannot query the array layout of a structure field or property.

Tips

- Outside of code generation and simulation, `coder.isRowMajor` is always false.
- If `coder.isRowMajor` always resolves to true for your code, other branches in the if statement are ignored by the code generator. Otherwise, one instance of the current function is created for each array layout.

See Also

`coder.ceval` | `coder.columnMajor` | `coder.isColumnMajor` | `coder.rowMajor`

Topics

“Interface with Row-Major Data in MATLAB Function Block”
“Specify Array Layout in Functions and Classes”
“Code Generation of Matrices and Arrays” (Simulink Coder)

Introduced in R2018a
**coder.load**

Load compile-time constants from MAT-file or ASCII file into caller workspace

**Syntax**

```matlab
S = coder.load(filename)
S = coder.load(filename,var1,...,varN)
S = coder.load(filename,'-regexp',expr1,...,exprN)
S = coder.load(filename,'-ascii')
S = coder.load(filename,'-mat')
S = coder.load(filename,'-mat','-regexp', expr1,...,exprN)
```

**Description**

*S* = `coder.load(filename)` loads compile-time constants from `filename`.

- If `filename` is a MAT-file, then `coder.load` loads variables from the MAT-file into a structure array.
- If `filename` is an ASCII file, then `coder.load` loads data into a double-precision array.

`coder.load` loads data at code generation time, also referred to as *compile time*. If you change the content of `filename` after you generate code, the change is not reflected in the behavior of the generated code.

*S* = `coder.load(filename,var1,...,varN)` loads only the specified variables from the MAT-file `filename`.

*S* = `coder.load(filename,'-regexp',expr1,...,exprN)` loads only the variables that match the specified regular expressions.

*S* = `coder.load(filename,'-ascii')` treats `filename` as an ASCII file, regardless of the file extension.
\[ S = \text{coder.load}(\text{filename}, \text{'-mat'}) \] treats \text{filename} as a MAT-file, regardless of the file extension.

\[ S = \text{coder.load}(\text{filename}, \text{'-mat'}, \text{var1}, \ldots, \text{varN}) \] treats \text{filename} as a MAT-file and loads only the specified variables from the file.

\[ S = \text{coder.load}(\text{filename}, \text{'-mat'}, \text{'-regexp'}, \text{expr1}, \ldots, \text{exprN}) \] treats \text{filename} as a MAT-file and loads only the variables that match the specified regular expressions.

**Examples**

**Load compile-time constants from MAT-file**

Generate code for a function \texttt{edgeDetect1} which given a normalized image, returns an image where the edges are detected with respect to the threshold value. \texttt{edgeDetect1} uses \texttt{coder.load} to load the edge detection kernel from a MAT-file at compile time.

Save the Sobel edge-detection kernel in a MAT-file.

\[ k = \begin{bmatrix} 1 & 2 & 1; 0 & 0 & 0; -1 & -2 & -1 \end{bmatrix}; \]

\[ \text{save sobel.mat } k \]

Write the function \texttt{edgeDetect1}.

\begin{verbatim}
function edgeImage = edgeDetect1(originalImage, threshold) %#codegen
assert(all(size(originalImage) <= [1024 1024]));
assert(isa(originalImage, 'double'));
assert(isa(threshold, 'double'));

S = coder.load('sobel.mat','k');
H = conv2(double(originalImage),S.k, 'same');
V = conv2(double(originalImage),S.k','same');
E = sqrt(H.*H + V.*V);
edgeImage = uint8((E > threshold) * 255);
\end{verbatim}

Create a code generation configuration object for a static library.

\[ \text{cfg = coder.config('lib')}; \]

Generate a static library for \texttt{edgeDetect1}. 

2-128
codegen -report -config cfg edgeDetect1

codegen generates C code in the codegen\lib\edgeDetect1 folder.

**Load compile-time constants from ASCII file**

Generate code for a function `edgeDetect2` which given a normalized image, returns an image where the edges are detected with respect to the threshold value. `edgeDetect2` uses `coder.load` to load the edge detection kernel from an ASCII file at compile time.

Save the Sobel edge-detection kernel in an ASCII file.

k = [1 2 1; 0 0 0; -1 -2 -1];
save sobel.dat k -ascii

Write the function `edgeDetect2`.

```matlab
function edgeImage = edgeDetect2(originalImage, threshold) %#codegen
    assert(all(size(originalImage) <= [1024 1024]));
    assert(isa(originalImage, 'double'));
    assert(isa(threshold, 'double'));

    k = coder.load('sobel.dat');
    H = conv2(double(originalImage),k, 'same');
    V = conv2(double(originalImage),k,'same');
    E = sqrt(H.*H + V.*V);
    edgeImage = uint8((E > threshold) * 255);

Create a code generation configuration object for a static library.

cfg = coder.config('lib');

Generate a static library for `edgeDetect2`.

codegen -report -config cfg edgeDetect2
```
codegen generates C code in the `codegen\lib\edgeDetect2` folder.

**Input Arguments**

`filename` — Name of file  
character vector | string scalar

Name of file. `filename` must be a compile-time constant.

`filename` can include a file extension and a full or partial path. If `filename` has no extension, `load` looks for a file named `filename.mat`. If `filename` has an extension other than `.mat`, `load` treats the file as ASCII data.

ASCII files must contain a rectangular table of numbers, with an equal number of elements in each row. The file delimiter (the character between elements in each row) can be a blank, comma, semicolon, or tab character. The file can contain MATLAB comments (lines that begin with a percent sign, %).

Example: `'myFile.mat'`

`var1,...,varN` — Names of variables to load  
character vector | string scalar

Names of variables, specified as one or more character vectors or string scalars. Each variable name must be a compile-time constant. Use the `*` wildcard to match patterns.

Example: `coder.load('myFile.mat','A*')` loads all variables in the file whose names start with `A`.

`expr1,...,exprN` — Regular expressions indicating which variables to load  
character vector | string scalar

Regular expressions indicating which variables to load specified as one or more character vectors or string scalars. Each regular expression must be a compile-time constant.

Example: `coder.load('myFile.mat', '-regexp', '^A')` loads only variables whose names begin with `A`. 
Output Arguments

S — Loaded variables or data
structure array | m-by-n array

If filename is a MAT-file, S is a structure array.

If filename is an ASCII file, S is an m-by-n array of type double. m is the number of lines in the file and n is the number of values on a line.

Limitations

• Arguments to coder.load must be compile-time constants.
• The output S must be the name of a structure or array without any subscripting. For example, S(i) = coder.load('myFile.mat') is not allowed.
• You cannot use save to save workspace data to a file inside a function intended for code generation. The code generator does not support the save function. Furthermore, you cannot use coder.extrinsic with save. Prior to generating code, you can use save to save workspace data to a file.

Tips

• coder.load loads data at compile time, not at run time. If you are generating MEX code or code for Simulink simulation, you can use the MATLAB function load to load run-time values.
• If the MAT-file contains unsupported constructs, use coder.load(filename, var1, ..., varN) to load only the supported constructs.
• If you generate code in a MATLAB Coder project, the code generator practices incremental code generation for the coder.load function. When the MAT-file or ASCII file used by coder.load changes, the software rebuilds the code.

See Also
matfile | regexp | save
Topics
“Regular Expressions” (MATLAB)

Introduced in R2013a
coder.nullcopy

Package: coder

Declare uninitialized variables in code generation

Syntax

X = coder.nullcopy(A)

Description

X = coder.nullcopy(A) copies type, size, and complexity of A to X, but does not copy element values. The function preallocates memory for X without incurring the overhead of initializing memory. In code generation, the coder.nullcopy function declares uninitialized variables. In MATLAB, coder.nullcopy returns the input such that X is equal to A.

If X is a structure containing variable-sized arrays, then you must assign the size of each array. coder.nullcopy does not copy sizes of arrays or nested arrays from its argument to its result.

Note Before you use X in a function or a program, ensure that the data in X is completely initialized. Declaring a variable through coder.nullcopy without assigning all the elements of the variable results in nondeterministic program behavior. For more information, see “How to Eliminate Redundant Copies by Defining Uninitialized Variables”.

Examples

Declare Variables for Optimized Initialization

Declare variable X as a 1-by-5 vector of real doubles without performing an unnecessary initialization:
function X = foo %#codegen

N = 5;
X = coder.nullcopy(zeros(1,N));
for i = 1:N
    if mod(i,2) == 0
        X(i) = i;
    else
        X(i) = 0;
    end
end

Using coder.nullcopy with zeros lets you specify the size of vector X without initializing each element to zero.

Input Arguments

A — Variable to copy
scalar | vector | matrix | multidimensional array

Variable to copy, specified as a scalar, vector, matrix, or multidimensional array.

Example: coder.nullcopy(A);

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | logical | char | string
Complex Number Support: Yes

Limitations

You cannot use coder.nullcopy on sparse matrices, structures, cell arrays, or classes that contain sparse matrices.

See Also

Topics
“Eliminate Redundant Copies of Variables in Generated Code”
Introduced in R2011a
**coder.opaque**

Declare variable in generated code

**Syntax**

\[
y = \text{coder.opaque}(	ext{type})
\]
\[
y = \text{coder.opaque}(	ext{type}, \text{value})
\]
\[
y = \text{coder.opaque}(\_\_, \text{'Size'}, \text{Size})
\]
\[
y = \text{coder.opaque}(\_\_, \text{'HeaderFile'}, \text{HeaderFile})
\]

**Description**

\[
y = \text{coder.opaque}(	ext{type})
\] declares a variable \( y \) with the specified type and no initial value in the generated code.

- \( y \) can be a variable or a structure field.
- MATLAB code cannot set or access \( y \), but external C functions can accept \( y \) as an argument.
- \( y \) can be an:
  - Argument to \text{coder.rref}, \text{coder.wref}, or \text{coder.ref}
  - Input or output argument to \text{coder.ceval}
  - Input or output argument to a user-written MATLAB function
  - Input to a subset of MATLAB toolbox functions supported for code generation
- Assignment from \( y \) declares another variable with the same type in the generated code. For example:

\[
y = \text{coder.opaque}('\text{int}')
\]
\[
z = y;
\]

declares a variable \( z \) of type \text{int} in the generated code.

- You can assign \( y \) from another variable declared using either \text{coder.opaque} or assignment from a variable declared using \text{coder.opaque}. The variables must have identical types.
• You can compare `y` to another variable declared using either `coder.opaque` or assignment from a variable declared using `coder.opaque`. The variables must have identical types.

\[
y = \text{coder.opaque}(\text{type},\text{value})\text{ specifies the type and initial value of } y.\]

\[
y = \text{coder.opaque}(\_\_\_,'\text{Size}',\text{Size})\text{ specifies the size, in bytes, of } y. \text{ You can specify the size with any of the previous syntaxes.}\]

\[
y = \text{coder.opaque}(\_\_\_,'\text{HeaderFile}',\text{HeaderFile})\text{ specifies the header file that contains the type definition. The code generator produces the } \#\text{include statement for the header file where the statement is required in the generated code. You can specify the header file with any of the previous syntaxes.}\]

### Examples

#### Declare Variable Specifying Initial Value

Generate code for a function `valtest` which returns 1 if the call to `myfun` is successful. This function uses `coder.opaque` to declare a variable `x1` with type `int` and initial value 0. The assignment `x2 = x1` declares `x2` to be a variable with the type and initial value of `x1`.

Write a function `valtest`.

```matlab
function y = valtest  
%codegen  
%declare x1 to be an integer with initial value '0'
x1 = coder.opaque('int','0');  
%Declare x2 to have same type and initial value as x1  
x2 = x1;  
x2 = coder.ceval('myfun');  
%test the result of call to 'myfun' by comparing to value of x1  
if x2 == x1  
    y = 0;  
else  
    y = 1;  
end
```
Declare Variable Specifying Initial Value and Header File

Generate code for a MATLAB function filetest which returns its own source code using fopen/fread/fclose. This function uses coder.opaque to declare the variable that stores the file pointer used by fopen/fread/fclose. The call to coder.opaque declares the variable f with type FILE *, initial value NULL, and header file <stdio.h>.

Write a MATLAB function filetest.

```matlab
function buffer = filetest
%#codegen
% Declare 'f' as an opaque type 'FILE *' with initial value 'NULL'
% Specify the header file that contains the type definition of 'FILE *';

f = coder.opaque('FILE *', 'NULL','HeaderFile','<stdio.h>');
% Open file in binary mode
f = coder.ceval('fopen', cstring('filetest.m'), cstring('rb'));
% Read from file until end of file is reached and put
% contents into buffer
n = int32(1);
i = int32(1);
buffer = char(zeros(1,8192));
while n > 0
    % By default, MATLAB converts constant values
    % to doubles in generated code
    % so explicit type conversion to int32 is inserted.
    n = coder.ceval('fread', coder.ref(buffer(i)), int32(1), ...
                    int32(numel(buffer)), f);
    i = i + n;
end
coder.ceval('fclose',f);

buffer = strip_cr(buffer);

% Put a C termination character '\0' at the end of MATLAB character vector
function y = cstring(x)
    y = [x char(0)];

% Remove all character 13 (CR) but keep character 10 (LF)
function buffer = strip_cr(buffer)
    j = 1;
    for i = 1:numel(buffer)
        if buffer(i) == char(13)
            buffer(j) = buffer(i);
            j = j + 1;
        end
    end
```

2-138
Compare Variables Declared Using coder.opaque

Compare variables declared using coder.opaque to test for successfully opening a file.

Use coder.opaque to declare a variable null with type FILE * and initial value NULL.

```
null = coder.opaque('FILE *', 'NULL', 'HeaderFile', '<stdio.h>');
```

Use assignment to declare another variable fttmp with the same type and value as null.

```
ftmp = null;
ftmp = coder.ceval('fopen', ['testfile.txt', char(0)], ['r', char(0)]);
```

Compare the variables.

```
if ftmp == null
    %error condition
end
```

Cast to and from Types of Variables Declared Using coder.opaque

This example shows how to cast to and from types of variables that are declared using coder.opaque. The function castopaque calls the C run-time function strncmp to compare at most n characters of the strings s1 and s2. n is the number of characters in the shorter of the strings. To generate the correct C type for the strncmp input nsizet, the function casts n to the C type size_t and assigns the result to nsizet. The function uses coder.opaque to declare nsizet. Before using the output retval from strncmp, the function casts retval to the MATLAB type int32 and stores the results in y.

Write this MATLAB function:

```
function y = castopaque(s1,s2)

%  <0 - the first character that does not match has a lower value in s1 than in s2
%  0 - the contents of both strings are equal
%  >0 - the first character that does not match has a greater value in s1 than in s2
%  
%#codegen
```
coder.cinclude('<string.h>);
n = min(numel(s1), numel(s2));

% Convert the number of characters to compare to a size_t
nsizet = cast(n,'like',coder.opaque('size_t','0'));

% The return value is an int
retval = coder.opaque('int');
retval = coder.ceval('strncmp', cstr(s1), cstr(s2), nsizet);

% Convert the opaque return value to a MATLAB value
y = cast(retval, 'int32');

%-----------------
function sc = cstr(s)
% NULL terminate a MATLAB character vector for C
sc = [s, char(0)];

Generate the MEX function.
codegen castopaque -args {blanks(3), blanks(3)} -report

Call the MEX function with inputs 'abc' and 'abc'.
castopaque_mex('abc','abc')
ans =

    0

The output is 0 because the strings are equal.

Call the MEX function with inputs 'abc' and 'abd'.
castopaque_mex('abc','abd')
ans =

    -1

The output is -1 because the third character d in the second string is greater than the third character c in the first string.

Call the MEX function with inputs 'abd' and 'abc'.

2-140
castopaque_mex('abd','abc')
ans =
 1

The output is 1 because the third character d in the first string is greater than the third character c in the second string.

In the MATLAB workspace, you can see that the type of y is int32.

**Declare Variable Specifying Initial Value and Size**

Declare y to be a 4-byte integer with initial value 0.

```matlab
y = coder.opaque('int','0', 'Size', 4);
```

**Input Arguments**

- **type** — Type of variable
  character vector | string scalar

Type of variable in generated code. type must be a compile-time constant. The type must be a:

- Built-in C data type or a type defined in a header file
- C type that supports copy by assignment
- Legal prefix in a C declaration

Example: 'FILE *'

- **value** — Initial value of variable
  character vector | string scalar

Initial value of variable in generated code. value must be a compile-time constant. Specify a C expression not dependent on MATLAB variables or functions.

If you do not provide the initial value in value, initialize the value of the variable before using it. To initialize a variable declared using coder.opaque:
• Assign a value from another variable with the same type declared using either \texttt{coder.opaque} or assignment from a variable declared using \texttt{coder.opaque}.

• Assign a value from an external C function.

• Pass the address of the variable to an external function using \texttt{coder.wref}.

Specify a value that has the type that \texttt{type} specifies. Otherwise, the generated code can produce unexpected results.

Example: \texttt{'NULL'}

\textbf{Size — Size of variable}

\texttt{integer}

Number of bytes for the variable in the generated code, specified as an integer. If you do not specify the size, the size of the variable is 8 bytes.

Data Types: \texttt{single} | \texttt{double} | \texttt{int8} | \texttt{int16} | \texttt{int32} | \texttt{int64} | \texttt{uint8} | \texttt{uint16} | \texttt{uint32} | \texttt{uint64}

\textbf{HeaderFile — Name of header file}

\texttt{character vector} | \texttt{string scalar}

Name of header file that contains the definition of \texttt{type}. \texttt{HeaderFile} must be a compile-time constant.

For a system header file, use angle brackets.

Example: \texttt{'<stdio.h>}' generates \texttt{#include <stdio.h>}

For an application header file, use double quotes.

Example: \texttt{"foo.h"} generates \texttt{#include "foo.h"}

If you omit the angle brackets or double quotes, the code generator produces double quotes.

Example: \texttt{'foo.h'} generates \texttt{#include "foo.h"}

Specify the include path in the build configuration parameters.

Example: \texttt{cfg.CustomInclude = 'c:\myincludes'}
Tips

- Specify a value that has the type that `type` specifies. Otherwise, the generated code can produce unexpected results. For example, the following `coder.opaque` declaration can produce unexpected results.

```matlab
y = coder.opaque('int', '0.2')
```

- `coder.opaque` declares the type of a variable. It does not instantiate the variable. You can instantiate a variable by using it later in the MATLAB code. In the following example, assignment of `fp1` from `coder.ceval` instantiates `fp1`.

```matlab
% Declare fp1 of type FILE *
fpl = coder.opaque('FILE *');
%Create the variable fp1
fp1 = coder.ceval('fopen', ['testfile.txt', char(0)], ['r', char(0)]);
```

- In the MATLAB environment, `coder.opaque` returns the value specified in `value`. If `value` is not provided, it returns an empty character vector.

- You can compare variables declared using either `coder.opaque` or assignment from a variable declared using `coder.opaque`. The variables must have identical types. The following example demonstrates how to compare these variables. “Compare Variables Declared Using coder.opaque” on page 2-139

- To avoid multiple inclusions of the same header file in generated code, enclose the header file in the conditional preprocessor statements `#ifndef` and `#endif`. For example:

```matlab
#ifndef MyHeader_h
#define MyHeader_h
<body of header file>
#endif
```

- You can use the MATLAB `cast` function to cast a variable to or from a variable that is declared using `coder.opaque`. Use `cast` with `coder.opaque` only for numeric types.

To cast a variable declared by `coder.opaque` to a MATLAB type, you can use the `B = cast(A, type)` syntax. For example:

```matlab
x = coder.opaque('size_t', '0');
x1 = cast(x, 'int32');
```

You can also use the `B = cast(A, 'like', p)` syntax. For example:
x = coder.opaque('size_t','0');
x1 = cast(x, 'like', int32(0));

To cast a MATLAB variable to the type of a variable declared by coder.opaque, you must use the B = cast(A,'like',p) syntax. For example:

x = int32(12);
x1 = coder.opaque('size_t', '0');
x2 = cast(x, 'like', x1));

Use cast with coder.opaque to generate the correct data types for:

- Inputs to C/C++ functions that you call using coder.ceval.
- Variables that you assign to outputs from C/C++ functions that you call using coder.ceval.

Without this casting, it is possible to receive compiler warnings during code generation.

**See Also**
coder.ceval | coder.ref | coder.rref | coder.wref

**Topics**
“Integrate C Code Using the MATLAB Function Block”

**Introduced in R2011a**
**coder.ref**

Indicate data to pass by reference

**Syntax**

coder.ref(arg)

**Description**

coder.ref(arg) indicates that arg is an expression or variable to pass by reference to an external C/C++ function. Use coder.ref inside a coder.ceval call only. The C/C++ function can read from or write to the variable passed by reference. Use a separate coder.ref construct for each argument that you pass by reference to the function.

See also coder.rref and coder.wref.

**Examples**

**Pass Scalar Variable by Reference**

Consider the C function addone that returns the value of an input plus one:

```c
double addone(double* p) {
    return *p + 1;
}
```

The C function defines the input variable p as a pointer to a double.

Pass the input by reference to addone:

```c
...  
y = 0;
u = 42;
```
y = coder.ceval('addone', coder.ref(u));
...

Pass Multiple Arguments by Reference
...
u = 1;
v = 2;
y = coder.ceval('my_fcn', coder.ref(u), coder.ref(v));
...

Pass Class Property by Reference
...
x = myClass;
x.prop = 1;
coder.ceval('foo', coder.ref(x.prop));
...

Pass a Structure by Reference

To indicate that the structure type is defined in a C header file, use coder.cstructname.

Suppose that you have the C function incr_struct. This function reads from and writes to the input argument.

#include "MyStruct.h"

void incr_struct(struct MyStruct *my_struct)
{
    my_struct->f1 = my_struct->f1 + 1;
    my_struct->f2 = my_struct->f2 + 1;
}

The C header file, MyStruct.h, defines a structure type named MyStruct:

#ifndef MYSTRUCT
#define MYSTRUCT
#define MYSTRUCT

typedef struct MyStruct
{

double f1;
    double f2;
} MyStruct;

void incr_struct(struct MyStruct *my_struct);

In your MATLAB function, pass a structure by reference to incr_struct. To indicate that the structure type for s has the name MyStruct that is defined in the C header file MyStruct.h, use coder.cstructname.

function y = foo
    %#codegen
    y = 0;
    coder.updateBuildInfo('addSourceFiles','incr_struct.c');

    s = struct('f1',1,'f2',2);
    coder.cstructname(s,'MyStruct','extern','HeaderFile','MyStruct.h');
    coder.ceval('incr_struct', coder.ref(s));

To generate standalone library code, enter:

codegen -config:lib foo -report

**Pass Structure Field by Reference**

... 
s = struct('s1', struct('a', [0 1]));
coder.ceval('foo', coder.ref(s.s1.a));
...

You can also pass an element of an array of structures:

... 
c = repmat(struct('u',magic(2)),1,10);
b = repmat(struct('c',c),3,6);
a = struct('b',b);
coder.ceval('foo', coder.ref(a.b(3,4).c(2).u));
...
Input Arguments

arg — Argument to pass by reference
scalar variable | array | element of an array | structure | structure field | object property

Argument to pass by reference to an external C/C++ function. The argument cannot be a class, a System object, a cell array, or an index into a cell array.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | logical | char | struct
Complex Number Support: Yes

Limitations

• You cannot pass these data types by reference:
  • Class or System object
  • Cell array or index into a cell array
  • If a property has a get method, a set method, or validators, or is a System object property with certain attributes, then you cannot pass the property by reference to an external function. See “Passing By Reference Not Supported for Some Properties”.

Tips

• If arg is an array, then coder.ref(arg) provides the address of the first element of the array. The coder.ref(arg) function does not contain information about the size of the array. If the C function must know the number of elements of your data, pass that information as a separate argument. For example:

coder.ceval('myFun',coder.ref(arg),int32(numel(arg)));

• When you pass a structure by reference to an external C/C++ function, use coder.cstructname to provide the name of a C structure type that is defined in a C header file.

• In MATLAB, coder.ref results in an error. To parameterize your MATLAB code so that it can run in MATLAB and in generated code, use coder.target.

• You can use coder.opaque to declare variables that you pass to and from an external C/C++ function.
See Also

coder.ceval | coder.cstructname | coder.opaque | coder.rref | coder.wref | numel

Topics

"Integrate C Code Using the MATLAB Function Block"

Introduced in R2011a
**coder.rowMajor**

Specify row-major array layout for a function or class

**Syntax**

coder.rowMajor

**Description**

coder.rowMajor specifies row-major array layout for the data used by the current function in generated code. When placed in a class constructor, coder.rowMajor specifies row-major layout for data used by the class.

**Examples**

**Specify Row-Major Array Layout for a Function**

Specify row-major array layout for a function by inserting coder.rowMajor into the function body.

Suppose that `myFunction` is the top-level function of your code. Your application requires you to perform matrix addition with row-major array layout and matrix multiplication with column-major layout.

```matlab
function S = myFunction(A,B)
%#codegen
% check to make sure inputs are valid
if size(A,1) ~= size(B,1) || size(A,2) ~= size(B,2)
    disp('Matrices must be same size.');
    return;
end
% make both matrices symmetric
B = B*B';
A = A*A';
```

2-150
% add matrices
S = addMatrix(A,B);
end

Write a function for matrix addition called addMatrix. Specify row-major layout for addMatrix by using coder.rowMajor.

function S = addMatrix(A,B)
%#codegen
S = zeros(size(A));
coder.rowMajor; % specify row-major array layout
S = A + B;
end

Generate code for myFunction. Use the codegen command.

codegen myFunction -args {ones(10,20),ones(10,20)} -config:lib -launchreport

The code generator produces code for addMatrix that uses row-major array layout. However, the matrix multiplication from the top-level function uses the default layout, column-major.

**Tips**

- To specify row-major array layout for all the functions in your generated code, use the codegen -rowmajor option.
- Other functions called from within a row-major function inherit the row-major specification. However, if one of the called functions has its own distinct coder.columnMajor call, the code generator changes the array layout accordingly. If a row-major function and a column-major function call the same function, which does not have its own array layout specification, the code generator produces a row-major version and column-major version of the function.
- coder.rowMajor is ignored outside of code generation and simulation.

**See Also**
coder.ceval | coder.columnMajor | coder.isColumnMajor | coder.isRowMajor

**Topics**
“Interface with Row-Major Data in MATLAB Function Block”
“Specify Array Layout in Functions and Classes”
“Code Generation of Matrices and Arrays” (Simulink Coder)

Introduced in R2018a
**coder.rref**

Indicate read-only data to pass by reference

**Syntax**

coder.rref(arg)

**Description**

coder.rref(arg) indicates that arg is a read-only expression or variable to pass by reference to an external C/C++ function. Use coder.rref only inside a coder.ceval call.

The coder.rref function can enable the code generator to optimize the generated code. Because the external function is assumed to not write to coder.rref(arg), the code generator can perform optimizations such as expression folding on assignments to arg that occur before and after the coder.ceval call. Expression folding is the combining of multiple operations into one statement to avoid the use of temporary variables and improve code performance.

**Note** The code generator assumes that the memory that you pass with coder.rref(arg) is read-only. To avoid unpredictable results, the C/C++ function must not write to this variable.

See also coder.ref and coder.wref.

**Examples**

**Pass Scalar Variable as a Read-Only Reference**

Consider the C function addone that returns the value of a constant input plus one:
double addone(const double* p) {
    return *p + 1;
}

The C function defines the input variable p as a pointer to a constant double.

Pass the input by reference to addone:

...  
y = 0;
u = 42;
y = coder.ceval('addone', coder.rref(u));
...

**Pass Multiple Arguments as a Read-Only Reference**

...
u = 1;
v = 2;
y = coder.ceval('my_fcn', coder.rref(u), coder.rref(v));
...

**Pass Class Property as a Read-Only Reference**

...
x = myClass;
x.prop = 1;
y = coder.ceval('foo', coder.rref(x.prop));
...

**Pass Structure as a Read-Only Reference**

To indicate that the structure type is defined in a C header file, use coder.cstructname.

Suppose that you have the C function use_struct. This function reads from the input argument but does not write to it.

#include "MyStruct.h"

double use_struct(const struct MyStruct *my_struct) {

return my_struct->f1 + my_struct->f2;
}

The C header file, MyStruct.h, defines a structure type named MyStruct:

#ifndef MYSTRUCT
#define MYSTRUCT

typedef struct MyStruct
{
   double f1;
   double f2;
} MyStruct;

double use_struct(const struct MyStruct *my_struct);
#endif

In your MATLAB function, pass a structure as a read-only reference to use_struct. To indicate that the structure type for s has the name MyStruct that is defined in the C header file MyStruct.h, use coder.cstructname.

function y = foo
%#codegen
y = 0;
coder.updateBuildInfo('addSourceFiles','use_struct.c');

s = struct('f1',1,'f2',2);
coder.cstructname(s,'MyStruct','extern','HeaderFile','MyStruct.h');
y = coder.ceval('use_struct', coder.rref(s));

To generate standalone library code, enter:

codegen -config:lib foo -report

Pass Structure Field as a Read-Only Reference

...  
s = struct('s1', struct('a', [0 1]));
y = coder.ceval('foo', coder.rref(s.s1.a));
...

You can also pass an element of an array of structures:
...  
c = repmat(struct('u',magic(2)),1,10);
b = repmat(struct('c',c),3,6);
a = struct('b',b);
coder.ceval('foo', coder.rref(a.b(3,4).c(2).u));
...

Input Arguments

arg — Argument to pass by reference
scalar variable | array | element of an array | structure | structure field | object property

Argument to pass by reference to an external C/C++ function. The argument cannot be a class, a System object, a cell array, or an index into a cell array.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | logical | char | struct
Complex Number Support: Yes

Limitations

• You cannot pass these data types by reference:
  • Class or System object
  • Cell array or index into a cell array
  • If a property has a get method, a set method, or validators, or is a System object property with certain attributes, then you cannot pass the property by reference to an external function. See “Passing By Reference Not Supported for Some Properties”.

Tips

• If arg is an array, then coder.rref(arg) provides the address of the first element of the array. The coder.rref(arg) function does not contain information about the size of the array. If the C function must know the number of elements of your data, pass that information as a separate argument. For example:

  coder.ceval('myFun',coder.rref(arg),int32(numel(arg)));
• When you pass a structure by reference to an external C/C++ function, use `coder.cstructname` to provide the name of a C structure type that is defined in a C header file.
• In MATLAB, `coder.rref` results in an error. To parametrize your MATLAB code so that it can run in MATLAB and in generated code, use `coder.target`.
• You can use `coder.opaque` to declare variables that you pass to and from an external C/C++ function.

See Also
`coder.ceval` | `coder.cstructname` | `coder.opaque` | `coder.ref` | `coder.wref`

Topics
“Integrate C Code Using the MATLAB Function Block”

Introduced in R2011a
coder.screener

Package: coder

Determine if function is suitable for code generation

Syntax

coder.screener(fcn)
coder.screener(fcn_1,...,fcn_n)

Description

coder.screener(fcn) analyzes the entry-point MATLAB function, fcn. It identifies unsupported functions and language features as code generation compliance issues. It displays the code generation compliance issues in a report. If fcn calls other functions directly or indirectly that are not MathWorks® functions, coder.screener analyzes these functions. It does not analyze MathWorks functions. It is possible that coder.screener does not detect all code generation issues. Under certain circumstances, it is possible that coder.screener reports false errors.

coder.screener does not report functions that the code generator automatically treats as extrinsic. Examples of such functions are plot, disp, and figure. See “Extrinsic Functions”.

coder.screener(fcn_1,...,fcn_n) analyzes entry-point functions (fcn_1,...,fcn_n).

Examples

Identify Unsupported Functions

The coder.screener function identifies calls to functions that are not supported for code generation. It checks both the entry-point function, foo1, and the function foo2 that foo1 calls.
Write the function `foo2` and save it in the file `foo2.m`.

```matlab
function out = foo2(in)
    out = eval(in);
end
```

Write the function `foo1` that calls `foo2`. Save `foo1` in the file `foo1.m`.

```matlab
function out = foo1(in)
    out = foo2(in);
    disp(out);
end
```

Analyze `foo1`.

```matlab
coder.screener('foo1')
```

The code generation readiness report displays a summary of the unsupported MATLAB function calls. The function `foo2` calls one unsupported MATLAB function.
In the report, click the **Code Structure** tab and select the **Show MATLAB functions** check box.

This tab displays a pie chart showing the relative size of each file and how suitable each file is for code generation. In this case, the report:
• Colors `foo1.m` green to indicate that it is suitable for code generation.
• Colors `foo2.m` yellow to indicate that it requires significant changes.
• Assigns `foo1.m` a code generation readiness score of 4 and `foo2.m` a score of 3. The score is based on a scale of 1–5. 1 indicates that significant changes are required; 5 indicates that the code generation readiness tool does not detect issues.
• Displays a call tree.
The report Summary tab indicates that foo2.m contains one call to the eval function, which code generation does not support. To generate a MEX function for foo2.m, modify the code to make the call to eval extrinsic.

```
function out = foo2(in)
    coder.extrinsic('eval');
```
out = eval(in);
end

Rerun the code generation readiness tool.
coder.screener('foo1')

The report no longer flags that code generation does not support the eval function. When you generate a MEX function for foo1, the code generator dispatches eval to MATLAB for execution. For standalone code generation, the code generator does not generate code for eval.

**Identify Unsupported Data Types**

The coder.screener function identifies MATLAB data types that code generation does not support.

Write the function myfun that contains a MATLAB calendar duration array data type.

```matlab
function out = myfun1(A)
out = calyears(A);
end
```

Analyze myfun.
coder.screener('myfun1');

The code generation readiness report indicates that the calyears data type is not supported for code generation.
The report assigns myfun1 a code readiness score of 3. Before generating code, you must fix the reported issues.

**Input Arguments**

`fcn` — Name of entry-point function  
character vector | string scalar  
Name of entry-point MATLAB function that you want to analyze. Specify as a character vector or a string scalar.  
Example: `coder.screener('myfun1');`  
Data Types: `char` | `string`

`fcn_1,...,fcn_n` — List of names of entry-point functions  
character vector | string scalar  
Comma-separated list of names of entry-point MATLAB functions that you want to analyze. Specify as character vectors or string scalars.
Example: coder.screener('myfun1','myfun2');

Data Types: char | string

**Tips**

- Before using `coder.screener`, fix issues that the Code Analyzer identifies.
- Before generating code, use `coder.screener` to check that a function is suitable for code generation. Fix all the issues that it detects.
- `coder.screener` does not report functions that the code generator automatically treats as extrinsic. Examples of such functions are `plot`, `disp`, and `figure`. See “Extrinsic Functions”.

**Alternatives**

- “Run the Code Generation Readiness Tool From the Current Folder Browser”

**See Also**

**Topics**

“Functions and Objects Supported for C/C++ Code Generation”
“Code Generation Readiness Tool”

**Introduced in R2012b**
**coder.target**

Determine if code generation target is specified target

**Syntax**

```
tf = coder.target(target)
```

**Description**

`tf = coder.target(target)` returns true (1) if the code generation target is `target`. Otherwise, it returns false (0).

If you generate code for MATLAB classes, MATLAB computes class initial values at class loading time before code generation. If you use `coder.target` in MATLAB class property initialization, `coder.target('MATLAB')` returns true.

**Examples**

**Use coder.target to Parametrize a MATLAB Function**

Parametrize a MATLAB function so that it works in MATLAB or in generated code. When the function runs in MATLAB, it calls the MATLAB function `myabsval`. The generated code, however, calls a C library function `myabsval`.

Write a MATLAB function `myabsval`.

```
function y = myabsval(u)
    y = abs(u);
```

Generate a C static library for `myabsval`, using the `-args` option to specify the size, type, and complexity of the input parameter.

```
codegen -config:lib myabsval -args {0.0}
```
The `codegen` function creates the library file `myabsval.lib` and header file `myabsval.h` in the folder `\codegen\lib\myabsval`. (The library file extension can change depending on your platform.) It generates the functions `myabsval_initialize` and `myabsval_terminate` in the same folder.

Write a MATLAB function to call the generated C library function using `coder.ceval`.

```matlab
function y = callmyabsval(y)
%#codegen
% Check the target. Do not use coder.ceval if callmyabsval is
% executing in MATLAB
if coder.target('MATLAB')
    % Executing in MATLAB, call function myabsval
    y = myabsval(y);
else
    % add the required include statements to generated function code
    coder.updateBuildInfo('addIncludePaths', '$(START_DIR)\codegen\lib\myabsval');
    coder.cinclude('myabsval_initialize.h');
    coder.cinclude('myabsval.h');
    coder.cinclude('myabsval_terminate.h');

    % Executing in the generated code.
    % Call the initialize function before calling the
    % C function for the first time
    coder.ceval('myabsval_initialize');

    % Call the generated C library function myabsval
    y = coder.ceval('myabsval', y);

    % Call the terminate function after
    % calling the C function for the last time
    coder.ceval('myabsval_terminate');
end
```

Generate the MEX function `callmyabsval_mex`. Provide the generated library file at the command line.

```
codegen -config:mex callmyabsval codegen\lib\myabsval\myabsval.lib -args {-2.75}
```

Rather than providing the library at the command line, you can use `coder.updateBuildInfo` to specify the library within the function. Use this option to preconfigure the build. Add this line to the `else` block:

```
coder.updateBuildInfo('addLinkObjects', 'myabsval.lib', '$(START_DIR)\codegen\lib\myabsval');
```
Run the MEX function `callmyabsval_mex` which calls the library function `myabsval`.

```matlab
callmyabsval_mex(-2.75)
ans =
    2.7500
```

Call the MATLAB function `callmyabsval`.

```matlab
callmyabsval(-2.75)
ans =
    2.7500
```

The `callmyabsval` function exhibits the desired behavior for execution in MATLAB and in code generation.

### Input Arguments

**target — code generation target**

A character vector: 'MATLAB', 'MEX', 'Sfun', 'Rtw', 'HDL', or 'Custom'.

- 'MATLAB': Running in MATLAB (not generating code)
- 'MEX': Generating a MEX function
- 'Sfun': Simulating a Simulink model
- 'Rtw': Generating a LIB, DLL, or EXE target
- 'HDL': Generating an HDL target
- 'Custom': Generating a custom target

```matlab
Example: tf = coder.target('MATLAB')
Example: tf = coder.target("MATLAB")
```
See Also

coder.BuildConfig | coder.ExternalDependency | coder.ceval |
coder.cinclude | coder.updateBuildInfo

Topics

“Integrate C Code Using the MATLAB Function Block”

Introduced in R2011a
**coder.unroll**

Unroll for-loop by making a copy of the loop body for each loop iteration

**Syntax**

coder.unroll()
coder.unroll(flag)

**Description**

coder.unroll() unrolls a for-loop. The coder.unroll call must be on a line by itself immediately preceding the for-loop that it unrolls.

Instead of producing a for-loop in the generated code, loop unrolling produces a copy of the for-loop body for each loop iteration. In each iteration, the loop index becomes constant. To unroll a loop, the code generator must be able to determine the bounds of the for-loop.

For small, tight loops, unrolling can improve performance. However, for large loops, unrolling can increase code generation time significantly and generate inefficient code.

coder.unroll is ignored outside of code generation.

coder.unroll(flag) unrolls a for-loop if flag is true. flag is evaluated at code generation time. The coder.unroll call must be on a line by itself immediately preceding the for-loop that it unrolls.

**Examples**

**Unroll a for-loop**

To produce copies of a for-loop body in the generated code, use coder.unroll.
In one file, write the entry-point function `call_getrand` and a local function `getrand`. `getrand` unrolls a `for`-loop that assigns random numbers to an `n`-by-1 array. `call_getrand` calls `getrand` with the value 3.

```matlab
function z = call_getrand
%#codegen
z = getrand(3);
end

function y = getrand(n)
coder.inline('never');
y = zeros(n, 1);
coder.unroll();
for i = 1:n
    y(i) = rand();
end
end
```

Generate a static library.

```bash
codegen -config:lib call_getrand -report
```

In the generated code, the code generator produces a copy of the `for`-loop body for each of the three loop iterations.

```c
static void getrand(double y[3])
{
    y[0] = b_rand();
    y[1] = b_rand();
    y[2] = b_rand();
}
```

**Control for-loop Unrolling with Flag**

Control loop unrolling by using `coder.unroll` with the flag argument.

In one file, write the entry-point function `call_getrand_unrollflag` and a local function `getrand_unrollflag`. When the number of loop iterations is less than 10, `getrand_unrollflag` unrolls the `for`-loop. `call_getrand` calls `getrand` with the value 50.

```matlab
function z = call_getrand_unrollflag
%#codegen
z = getrand_unrollflag(50);
```
function y = getrand_unrollflag(n)
coder.inline('never');
unrollflag = n < 10;
y = zeros(n, 1);
coder.unroll(unrollflag)
for i = 1:n
    y(i) = rand();
end
end

Generate a static library.

codegen -config:lib call_getrand_unrollflag -report

The number of iterations is not less than 10. Therefore, the code generator does not
unroll the for-loop. It produces a for-loop in the generated code.

static void getrand_unrollflag(double y[50])
{
    int i;
    for (i = 0; i < 50; i++) {
        y[i] = b_rand();
    }
}

**Use Legacy Syntax to Unroll for-Loop**

function z = call_getrand
%
#codegen
z = getrand(3);
end

function y = getrand(n)
coder.inline('never');
y = zeros(n, 1);
for i = coder.unroll(1:n)
    y(i) = rand();
end
end
Use Legacy Syntax to Control for-Loop Unrolling

```matlab
function z = call_getrand_unrollflag
%#codegen
z = getrand_unrollflag(50);
end

function y = getrand_unrollflag(n)
coder.inline('never');
unrollflag = n < 10;
y = zeros(n, 1);
for i = coder.unroll(1:n, unrollflag)
    y(i) = rand();
end
end
```

### Input Arguments

**flag** — Indicates whether to unroll the for-loop

*true (default) | false*

When `flag` is `true`, the code generator unrolls the for-loop. When `flag` is `false`, the code generator produces a for-loop in the generated code. `flag` is evaluated at code generation time.

### Tips

- Sometimes, the code generator unrolls a for-loop even though you do not use `coder.unroll`. For example, if a for-loop indexes into a heterogeneous cell array or into `varargin` or `varargout`, the code generator unrolls the loop. By unrolling the loop, the code generator can determine the value of the index for each loop iteration. The code generator uses heuristics to determine when to unroll a for-loop. If the heuristics fail to identify that unrolling is warranted, or if the number of loop iterations exceeds a limit, code generation fails. In these cases, you can force loop unrolling by using `coder.unroll`. See “Nonconstant Index into varargin or varargout in a for-Loop”.
See Also
coder.inline

Topics
“Unroll for-Loops”
“Nonconstant Index into varargin or varargout in a for-Loop”

Introduced in R2011a
**coder.updateBuildInfo**

Update build information object `RTW.BuildInfo`

**Syntax**

```matlab
coder.updateBuildInfo('addCompileFlags',options)  
coder.updateBuildInfo('addLinkFlags',options)  
coder.updateBuildInfo('addDefines',options)  
coder.updateBuildInfo( ___ ,group)  

coder.updateBuildInfo('addLinkObjects',filename,path)  
coder.updateBuildInfo('addLinkObjects',filename,path,priority, precompiled)  
coder.updateBuildInfo('addLinkObjects',filename,path,priority, precompiled,linkonly)  
coder.updateBuildInfo( ___ ,group)  

coder.updateBuildInfo('addNonBuildFiles',filename)  
coder.updateBuildInfo('addSourceFiles',filename)  
coder.updateBuildInfo('addIncludeFiles',filename)  
coder.updateBuildInfo( ___ ,path)  
coder.updateBuildInfo( ___ ,path,group)  

coder.updateBuildInfo('addSourcePaths',path)  
coder.updateBuildInfo('addIncludePaths',path)  
coder.updateBuildInfo( ___ ,group)  
```

**Description**

`coder.updateBuildInfo('addCompileFlags',options)` adds compiler options to the build information object.

`coder.updateBuildInfo('addLinkFlags',options)` adds link options to the build information object.
coder.updateBuildInfo('addDefines',options) adds preprocessor macro definitions to the build information object.

coder.updateBuildInfo(____,group) assigns a group name to options for later reference.

coder.updateBuildInfo('addLinkObjects',filename,path) adds a link object from a file to the build information object.

coder.updateBuildInfo('addLinkObjects',filename,path,priority,precompiled) specifies if the link object is precompiled.

coder.updateBuildInfo('addLinkObjects',filename,path,priority,precompiled,linkonly) specifies if the object is to be built before being linked or used for linking alone. If the object is to be built, it specifies if the object is precompiled.

coder.updateBuildInfo(____,group) assigns a group name to the link object for later reference.

coder.updateBuildInfo('addNonBuildFiles',filename) adds a nonbuild-related file to the build information object.

coder.updateBuildInfo('addSourceFiles',filename) adds a source file to the build information object.

coder.updateBuildInfo('addIncludeFiles',filename) adds an include file to the build information object.

coder.updateBuildInfo(____,path) adds the file from specified path.

coder.updateBuildInfo(____,path,group) assigns a group name to the file for later reference.

coder.updateBuildInfo('addSourcePaths',path) adds a source file path to the build information object.

coder.updateBuildInfo('addIncludePaths',path) adds an include file path to the build information object.

coder.updateBuildInfo(____,group) assigns a group name to the path for later reference.
Examples

Add Multiple Compiler Options

Add the compiler options -Zi and -Wall during code generation for function, func.

Anywhere in the MATLAB code for func, add the following line:

coder.updateBuildInfo('addCompileFlags','-Zi -Wall');

Generate code for func using the codegen command. Open the Code Generation Report.

codegen -config:lib -launchreport func

Add Source File Name

Add a source file to the project build information while generating code for a function, calc_factorial.

1. Write a header file fact.h that declares a C function factorial.

   double factorial(double x);

   fact.h will be included as a header file in generated code. This inclusion ensures that the function is declared before it is called.

   Save the file in the current folder.

2. Write a C file fact.c that contains the definition of factorial. factorial calculates the factorial of its input.

   #include "fact.h"

   double factorial(double x)
   {
     int i;
     double fact = 1.0;
     if (x == 0 || x == 1) {
       return 1.0;
     } else {
       for (i = 1; i <= x; i++) {
         fact *= i;
       }
       return fact;
     }
fact *= (double)i;
}
return fact;
}

fact.c is used as a source file during code generation.

Save the file in the current folder.

3 Write a MATLAB function calc_factorial that uses coder.ceval to call the external C function factorial.

Use coder.updateBuildInfo with option 'addSourceFiles' to add the source file fact.c to the build information. Use coder.cinclud to include the header file fact.h in the generated code.

```matlab
function y = calc_factorial(x) %#codegen
coder.cinclud('fact.h');
coder.updateBuildInfo('addSourceFiles', 'fact.c');
y = 0;
y = coder.ceval('factorial', x);
```

4 Generate code for calc_factorial using the codegen command.

```
codegen -config:dll -launchreport calc_factorial -args 0
```

Add Link Object

Add a link object LinkObj.lib to the build information while generating code for a function func. For this example, you must have a link object LinkObj.lib saved in a local folder, for example, c:\Link_Objects.

Anywhere in the MATLAB code for func, add the following lines:

```matlab
libPriority = '';
libPreCompiled = true;
libLinkOnly = true;
libName = 'LinkObj.lib';
libPath = 'c:\Link_Objects';
```
coder.updateBuildInfo('addLinkObjects', libName, libPath, ...
   libPriority, libPreCompiled, libLinkOnly);

Generate a MEX function for func using the codegen command. Open the Code Generation Report.

codegen -launchreport func

**Add Include Paths**

Add an include path to the build information while generating code for a function, adder. Include a header file, adder.h, existing on the path.

When header files do not reside in the current folder, to include them, use this method:

1. Write a header file mysum.h that contains the declaration for a C function mysum.
   
   ```
   double mysum(double, double);
   ```
   
   Save it in a local folder, for example c:\coder\myheaders.

2. Write a C file mysum.c that contains the definition of the function mysum.
   
   ```
   #include "mysum.h"
   double mysum(double x, double y)
   {
     return(x+y);
   }
   ```
   
   Save it in the current folder.

3. Write a MATLAB function adder that adds the path c:\coder\myheaders to the build information.

   Use coder.cinclude to include the header file mysum.h in the generated code.

   ```
   function y = adder(x1, x2) %#codegen
       coder.updateBuildInfo('addIncludePaths','c:\coder\myheaders');
       coder.updateBuildInfo('addSourceFiles','mysum.c');
       %Include the source file containing C function definition
       coder.cinclude('mysum.h');
       y = 0;
   ```
if coder.target('MATLAB')
    y = x1 + x2;
else
    y = coder.ceval('mySum', x1, x2);
end
end

Generate code for adder using the codegen command.

codegen -config:lib -launchreport adder -args {0,0}

Input Arguments

options — Build options
character vector | string scalar

Build options, specified as a character vector or string scalar. The value must be a compile-time constant.

Depending on the leading argument, options specifies the relevant build options to be added to the project’s build information.

<table>
<thead>
<tr>
<th>Leading Argument</th>
<th>Values in options</th>
</tr>
</thead>
<tbody>
<tr>
<td>'addCompileFlags'</td>
<td>Compiler options</td>
</tr>
<tr>
<td>'addLinkFlags'</td>
<td>Link options</td>
</tr>
<tr>
<td>'addDefines'</td>
<td>Preprocessor macro definitions</td>
</tr>
</tbody>
</table>

The function adds the options to the end of an option vector.

Example: coder.updateBuildInfo('addCompileFlags','-Zi -Wall')

group — Group name
character vector | string scalar

Name of user-defined group, specified as a character vector or string scalar. The value must be a compile-time constant.

The group option assigns a group name to the parameters in the second argument.
<table>
<thead>
<tr>
<th>Leading Argument</th>
<th>Second Argument</th>
<th>Parameters Named by group</th>
</tr>
</thead>
<tbody>
<tr>
<td>'addCompileFlags'</td>
<td>options</td>
<td>Compiler options</td>
</tr>
<tr>
<td>'addLinkFlags'</td>
<td>options</td>
<td>Link options</td>
</tr>
<tr>
<td>'addLinkObjects'</td>
<td>filename</td>
<td>Name of file containing linkable objects</td>
</tr>
<tr>
<td>'addNonBuildFiles'</td>
<td>filename</td>
<td>Name of nonbuild-related file</td>
</tr>
<tr>
<td>'addSourceFiles'</td>
<td>filename</td>
<td>Name of source file</td>
</tr>
<tr>
<td>'addSourcePaths'</td>
<td>path</td>
<td>Name of source file path</td>
</tr>
</tbody>
</table>

You can use group to:

- Document the use of specific parameters.
- Retrieve or apply multiple parameters together as one group.

**filename — File name**

character vector | string scalar

File name, specified as a character vector or string scalar. The value must be a compile-time constant.

Depending on the leading argument, filename specifies the relevant file to be added to the project’s build information.

<table>
<thead>
<tr>
<th>Leading Argument</th>
<th>File Specified by filename</th>
</tr>
</thead>
<tbody>
<tr>
<td>'addLinkObjects'</td>
<td>File containing linkable objects</td>
</tr>
<tr>
<td>'addNonBuildFiles'</td>
<td>Nonbuild-related file</td>
</tr>
<tr>
<td>'addSourceFiles'</td>
<td>Source file</td>
</tr>
</tbody>
</table>

The function adds the file name to the end of a file name vector.

Example: `coder.updateBuildInfo('addSourceFiles', 'fact.c')`

**path — Path name**

character vector | string scalar
Relative path name, specified as a character vector or string scalar. The value must be a compile-time constant.

Depending on the leading argument, path specifies the relevant path name to be added to the project’s build information. The function adds the path to the end of a path name vector.

<table>
<thead>
<tr>
<th>Leading Argument</th>
<th>Path Specified by path</th>
</tr>
</thead>
<tbody>
<tr>
<td>'addLinkObjects'</td>
<td>Path to linkable objects</td>
</tr>
<tr>
<td>'addNonBuildFiles'</td>
<td>Path to nonbuild-related files</td>
</tr>
<tr>
<td>'addSourceFiles', 'addSourcePaths'</td>
<td>Path to source files</td>
</tr>
</tbody>
</table>

The relative path starts from the build folder. If you have a function foo contained in the folder C:\myCode, and you generate MEX code by using:

codegen foo -report

then the build folder is C:\myCode\codegen\mex\foo. You can write the path from the build folder or write the path from the current working folder in which you generate code. Reference the current working folder by using the START_DIR macro. For example, suppose that your source file is contained in C:\myCode\mySrcDir, and you generate code from C:\myCode. Write the path as in these examples:

Example:
coder.updateBuildInfo('addSourceFiles','fact.c','..\..\..\..\mySrcDir')
Example: coder.updateBuildInfo('addSourceFiles','fact.c','$\{\text{START\_DIR}\}\mySrcDir')

**priority** — Relative priority of link object

Priority of link objects.

This feature applies only when several link objects are added. Currently, only a single link object file can be added for every coder.updateBuildInfo statement. Therefore, this feature is not available for use.

To use the succeeding arguments, include '' as a placeholder argument.

**precompiled** — Variable indicating if link objects are precompiled

logical value
Variable indicating if the link objects are precompiled, specified as a logical value. The value must be a compile-time constant.

If the link object has been prebuilt for faster compiling and linking and exists in a specified location, specify true. Otherwise, the MATLAB Coder build process creates the link object in the build folder.

If linkonly is set to true, this argument is ignored.

Data Types: logical

linkonly — Variable indicating if objects must be used for linking only
logical value

Variable indicating if objects must be used for linking only, specified as a logical value. The value must be a compile-time constant.

If you want that the MATLAB Coder build process must not build or generate rules in the makefile for building the specified link object, specify true. Instead, when linking the final executable, the process should just include the object. Otherwise, rules for building the link object are added to the makefile.

You can use this argument to incorporate link objects for which source files are not available.

If linkonly is set to true, the value of precompiled is ignored.

Data Types: logical

See Also
coder.ExternalDependency | coder.ceval | coder.cincluder | coder.ref | coder.rref | coder.target | coder.wref

Topics
“Build Process Customization” (MATLAB Coder)
“Integrate C Code Using the MATLAB Function Block”

Introduced in R2013b
coder.varsize

**Package:** coder

Declare variable-size data

**Syntax**

coder.varsize(varName1,...,varNameN)
coder.varsize(varName1,...,varNameN,ubounds)
coder.varsize(varName1,...,varNameN,ubounds,dims)

**Description**

coder.varsize(varName1,...,varNameN) declares that the variables named varName1,...,varNameN have a variable size. The declaration instructs the code generator to allow the variables to change size during execution of the generated code. With this syntax, you do not specify the upper bounds of the dimensions of the variables or which dimensions can change size. The code generator computes the upper bounds. All dimensions, except singleton dimensions on page 2-191, are allowed to change size.

Use `coder.varsize` according to these restrictions and guidelines:

- Use `coder.varsize` inside a MATLAB function intended for code generation.
- The `coder.varsize` declaration must precede the first use of a variable. For example:

  ```matlab
  ...
  x = 1;
  coder.varsize('x');
  disp(size(x));
  ...
  ```

- Use `coder.varsize` to declare that an output argument has a variable size or to address size mismatch errors. Otherwise, to define variable-size data, use the methods described in “Define Variable-Size Data for Code Generation”.
Note For MATLAB Function blocks, to declare variable-size input or output signals, use the Ports and Data Manager. See “Declare Variable-Size Inputs and Outputs”. If you provide upper bounds in a `coder.varsize` declaration, the upper bounds must match the upper bounds in the Ports and Data Manager.

For more restrictions and guidelines, see “Limitations” on page 2-189 and “Tips” on page 2-191.

`coder.varsize(varName1,...,varNameN,ubounds)` also specifies an upper bound for each dimension of the variables. All variables must have the same number of dimensions. All dimensions, except singleton dimensions on page 2-191, are allowed to change size.

`coder.varsize(varName1,...,varNameN,ubounds,dims)` also specifies an upper bound for each dimension of the variables and whether each dimension has a fixed size or a variable size. If a dimension has a fixed size, then the corresponding `ubound` element specifies the fixed size of the dimension. All variables have the same fixed-size dimensions and the same variable-size dimensions.

Examples

Address Size Mismatch Error by Using `coder.varsize`

After a variable is used (read), changing the size of the variable can cause a size mismatch error. Use `coder.varsize` to specify that the size of the variable can change.

Code generation for the following function produces a size mismatch error because `x = 1:10` changes the size of the second dimension of `x` after the line `y = size(x)` that uses `x`.

```matlab
function [x,y] = usevarsize(n)
  %#codegen
  x = 1;
  y = size(x);
  if n > 10
    x = 1:10;
  end
```

To declare that `x` can change size, use `coder.varsize`. 
function [x,y] = usevarsize(n)
  %#codegen
  x = 1;
  coder.varsize('x');
  y = size(x);
  if n > 10
    x = 1:10;
  end

If you remove the line `y = size(x)`, you no longer need the `coder.varsize` declaration because `x` is not used before its size changes.

**Declare Variable-Size Array with Upper Bounds**

Specify that `A` is a row vector whose second dimension has a variable size with an upper bound of 20.

```matlab
function fcn()
  ...
  coder.varsize('A',[1 20]);
  ...
end
```

When you do not provide `dims`, all dimensions, except singleton dimensions, have a variable size.

**Declare Variable-Size Array with a Mix of Fixed and Variable Dimensions**

Specify that `A` is an array whose first dimension has a fixed size of three and whose second dimension has a variable size with an upper bound of 20.

```matlab
function fcn()
  ...
  coder.varsize('A',[3 20], [0 1] );
  ...
end
```
Declare Variable-Size Structure Fields

If a structure field belongs to an array of structures, use colon (:) as the index expression to make the field variable-size for all elements of the array.

In this function, the statement `coder.varsize('data(:).values')` declares that the field values inside each element of data has a variable size.

```matlab
function y = varsize_field()
    %#codegen
    d = struct('values', zeros(1,0), 'color', 0);
    data = repmat(d, [3 3]);
    coder.varsize('data(:).values');

    for i = 1:numel(data)
        data(i).color = rand-0.5;
        data(i).values = 1:i;
    end

    y = 0;
    for i = 1:numel(data)
        if data(i).color > 0
            y = y + sum(data(i).values);
        end
    end
```

Declare Variable-Size Cell Array

Specify that cell array C has a fixed-size first dimension and variable-size second dimension with an upper bound of three. The `coder.varsize` declaration must precede the first use of C.

```matlab
... C = {1 [1 2]};
coder.varsize('C', [1 3], [0 1]);
y = C{1};
...
end
```

Without the `coder.varsize` declaration, C is a heterogeneous cell array whose elements have the same class and different sizes. With the `coder.varsize` declaration, C is a
homogeneous cell array whose elements have the same class and maximum size. The first dimension of each element is fixed at 1. The second dimension of each element has a variable size with an upper bound of 3.

Declare That a Cell Array Has Variable-Size Elements

Specify that the elements of cell array C are vectors with a fixed-size first dimension and variable-size second dimension with an upper bound of 5.

```matlab
... 
C = {1 2 3}; 
coder.varsize('C{:}', [1 5], [0 1]); 
C = {1, 1:5, 2:3}; 
...
```

Input Arguments

`varName1,...,varNameN` — Names of variables to declare as having a variable size
code vectors | string scalars

Names of variables to declare as having a variable size, specified as one or more character vectors or string scalars.
Example: `coder.varsize('x','y')`

`ubounds` — Upper bounds for array dimensions
[] (default) | vector of integer constants

Upper bounds for array dimensions, specified as a vector of integer constants.

When you do not specify `ubounds`, the code generator computes the upper bound for each variable. If the `ubounds` element corresponds to a fixed-size dimension, the value is the fixed size of the dimension.
Example: `coder.varsize('x','y',[1 2])`

`dims` — Indication of whether each dimension has a fixed size or a variable size
logical vector
Indication of whether each dimension has a fixed size or a variable size, specified as a logical vector. Dimensions that correspond to 0 or false in dims have a fixed size. Dimensions that correspond to 1 or true have a variable size.

When you do not specify dims, the dimensions have a variable size, except for the singleton dimensions.

Example: 
```plaintext
coder.varsize('x','y',[1 2], [0 1])
```

**Limitations**

- The `coder.varsize` declaration instructs the code generator to allow the size of a variable to change. It does not change the size of the variable. Consider this code:

  ```plaintext
  ...
  x = 7;
  coder.varsize('x', [1,5]);
  disp(size(x));
  ...
  ```

  After the `coder.varsize` declaration, `x` is still a 1-by-1 array. You cannot assign a value to an element beyond the current size of `x`. For example, this code produces a run-time error because the index 3 exceeds the dimensions of `x`.

  ```plaintext
  ...
  x = 7;
  coder.varsize('x', [1,5]);
  x(3) = 1;
  ...
  ```

- `coder.varsize` is not supported for a function input argument. Instead:
  - If the function is an entry-point function, specify that an input argument has a variable size by using `coder.typeof` at the command line. Alternatively, specify that an entry-point function input argument has a variable size by using the Define Input Types step of the app.
  - If the function is not an entry-point function, use `coder.varsize` in the calling function with the variable that is the input to the called function.

- For sparse matrices, `coder.varsize` drops upper bounds for variable-size dimensions.
- Limitations for using `coder.varsize` with cell arrays:
• A cell array can have a variable size only if it is homogeneous. When you use `coder.varsize` with a heterogeneous cell array, the code generator tries to make the cell array homogeneous. The code generator tries to find a class and maximum size that apply to all elements of the cell array. For example, consider the cell array \( c = \{1, [2 \ 3]\} \). Both elements can be represented by a double type whose first dimension has a fixed size of 1 and whose second dimension has a variable size with an upper bound of 2. If the code generator cannot find a common class and a maximum size, code generation fails. For example, consider the cell array \( c = \{'a',[2 \ 3]\} \). The code generator cannot find a class that can represent both elements because the first element is `char` and the second element is `double`.

• If you use the `cell` function to define a fixed-size cell array, you cannot use `coder.varsize` to specify that the cell array has a variable size. For example, this code causes a code generation error because \( x = \text{cell}(1,3) \) makes \( x \) a fixed-size,1-by-3 cell array.

```matlab
... 
x = cell(1,3);
coder.varsize('x',[1 5])
...
```

You can use `coder.varsize` with a cell array that you define by using curly braces. For example:

```matlab
... 
x = {1 \ 2 \ 3};
coder.varsize('x',[1 5])
...
```

• To create a variable-size cell array by using the `cell` function, use this code pattern:

```matlab
function mycell(n)
%#codegen
x = cell(1,n);
for i = 1:n
    x{i} = i;
end
end
```

See “Definition of Variable-Size Cell Array by Using cell”.

To specify upper bounds for the cell array, use `coder.varsize`.

```matlab
function mycell(n)
%#codegen
```
x = cell(1,n);
for i = 1:n
  x{i} = i;
coder.varsize('x',[1,20]);
end
end

More About

Singleton Dimension

Dimension for which size(A,dim) = 1.

Tips

- In a code generation report or a MATLAB Function report, a colon (:) indicates that a dimension has a variable size. For example, a size of 1x:2 indicates that the first dimension has a fixed size of one and the second dimension has a variable size with an upper bound of two.

- If you use coder.varsize to specify that the upper bound of a dimension is 1, by default, the dimension has a fixed size of 1. To specify that the dimension can be 0 (empty array) or 1, set the corresponding element of the dims argument to true. For example, this code specifies that the first dimension of x has a fixed size of 1 and the other dimensions have a variable size of 5.

  coder.varsize('x',[1,5,5])

  In contrast, this code specifies that the first dimension of x has an upper bound of 1 and has a variable size (can be 0 or 1).

  coder.varsize('x',[1,5,5],[1,1,1])

Note For a MATLAB Function block, you cannot specify that an input or output signal with size 1 has a variable size.

- If you use input variables or the result of a computation using input variables to specify the size of an array, it is declared as variable-size in the generated code. Do not re-use coder.varsize on the array, unless you also want to specify an upper bound for its size.
• If you do not specify upper bounds with a `coder.varsize` declaration and the code generator is unable to determine the upper bounds, the generated code uses dynamic memory allocation. Dynamic memory allocation can reduce the speed of generated code. To avoid dynamic memory allocation, specify the upper bounds by providing the `ubounds` argument.

See Also

Topics
“Code Generation for Variable-Size Arrays”
“Incompatibilities with MATLAB in Variable-Size Support for Code Generation”
“Avoid Duplicate Functions in Generated Code”

Introduced in R2011a
**coder.wref**

Indicate write-only data to pass by reference

**Syntax**

coder.wref(arg)

**Description**

coder.wref(arg) indicates that arg is a write-only expression or variable to pass by reference to an external C/C++ function. Use coder.wref only inside a coder.ceval call. This function enables the code generator to optimize the generated code by ignoring prior assignments to arg in your MATLAB code, because the external function is assumed to not read from the data. Write to all the elements of arg in your external code to fully initialize the memory.

**Note** The C/C++ function must fully initialize the memory referenced by coder.wref(arg). Initialize the memory by assigning values to every element of arg in your C/C++ code. If the generated code tries to read from uninitialized memory, it can cause undefined run-time behavior.

See also coder.ref and coder.rref.

**Examples**

**Pass Array by Reference as Write-Only**

Suppose that you have a C function init_array.

```c
void init_array(double* array, int numel) {
    for(int i = 0; i < numel; i++) {
        array[i] = 42;
    }
}
```
The C function defines the input variable array as a pointer to a double.

Call the C function init_array to initialize all elements of y to 42:

```matlab
... 
Y = zeros(5, 10);
coder.ceval('init_array', coder.wref(Y), int32(numel(Y)));
... 
```

**Pass Multiple Arguments as a Write-Only Reference**

```matlab
... 
U = zeros(5, 10);
V = zeros(5, 10);
coder.ceval('my_fcn', coder.wref(U), int32(numel(U)), coder.wref(V), int32(numel(V)));
... 
```

**Pass Class Property as a Write-Only Reference**

```matlab
... 
x = myClass;
x.prop = 1;
coder.ceval('foo', coder.wref(x.prop));
... 
```

**Pass Structure as a Write-Only Reference**

To indicate that the structure type is defined in a C header file, use coder.cstructname.

Suppose that you have the C function init_struct. This function writes to the input argument but does not read from it.

```c
#include "MyStruct.h"

void init_struct(struct MyStruct *my_struct)
{
    my_struct->f1 = 1;
    my_struct->f2 = 2;
}
```
The C header file, `MyStruct.h`, defines a structure type named `MyStruct`:

```c
#ifndef MYSTRUCT
#define MYSTRUCT

typedef struct MyStruct
{
    double f1;
    double f2;
} MyStruct;

void init_struct(struct MyStruct *my_struct);
#endif
```

In your MATLAB function, pass a structure as a write-only reference to `init_struct`. Use `coder.cstructname` to indicate that the structure type for `s` has the name `MyStruct` that is defined in the C header file `MyStruct.h`.

```matlab
function y = foo
    %#codegen
    y = 0;
    coder.updateBuildInfo('addSourceFiles','init_struct.c');

    s = struct('f1',1,'f2',2);
    coder.cstructname(s,'MyStruct','extern','HeaderFile','MyStruct.h');
    coder.ceval('init_struct', coder.wref(s));
```

To generate standalone library code, enter:

```
codegen -config:lib foo -report
```

### Pass Structure Field as a Write-Only Reference

```matlab
...  
s = struct('s1', struct('a', [0 1]));
coder.ceval('foo', coder.wref(s.s1.a));
...
```

You can also pass an element of an array of structures:

```matlab
...  
c = repmat(struct('u',magic(2)),1,10);
b = repmat(struct('c',c),3,6);
```
a = struct('b',b);
coder.ceval('foo', coder.wref(a.b(3,4).c(2).u));
...

Input Arguments

arg — Argument to pass by reference
scalar variable | array | element of an array | structure | structure field | object property

Argument to pass by reference to an external C/C++ function. The argument cannot be a class, a System object, a cell array, or an index into a cell array.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16
| uint32 | uint64 | logical | char | struct
Complex Number Support: Yes

Limitations

• You cannot pass these data types by reference:
  • Class or System object
  • Cell array or index into a cell array
• If a property has a get method, a set method, or validators, or is a System object property with certain attributes, then you cannot pass the property by reference to an external function. See “Passing By Reference Not Supported for Some Properties”.

Tips

• If arg is an array, then coder.wref(arg) provides the address of the first element of the array. The coder.wref(arg) function does not contain information about the size of the array. If the C function must know the number of elements of your data, pass that information as a separate argument. For example:

coder.ceval('myFun',coder.wref(arg),int32(numel(arg));
• When you pass a structure by reference to an external C/C++ function, use coder.cstructname to provide the name of a C structure type that is defined in a C header file.
• In MATLAB, `coder.wref` results in an error. To parametrize your MATLAB code so that it can run in MATLAB and in generated code, use `coder.target`.

• You can use `coder.opaque` to declare variables that you pass to and from an external C/C++ function.

See Also
`coder.ceval` | `coder.cstructname` | `coder.opaque` | `coder.ref` | `coder.rref`

Topics
“Integrate C Code Using the MATLAB Function Block”

Introduced in R2011a
createInputDataset

Generate dataset object for root-level Inport or bus element ports in model

Syntax

[inports_dataset] = createInputDataset(model)

Description

[inports_dataset] = createInputDataset(model) generates a Simulink.SimulationData.Dataset object from the root-level Inport blocks or bus element ports in a model. Signals in the generated dataset have the properties of the root inports and the corresponding ground values at model start and stop times. You can create timetable or timeseries objects for the time and values for signals for which you want to load data for simulation. The other signals use ground values.

Examples

Generate and Populate Dataset for Root-Level Inport Blocks

This example shows how to create a dataset with elements for the four root-level Inport blocks in a model. Use that dataset as a basis for creating a dataset to load signal data into the model.

The In1 block outputs a double, In2 and In3 each output a nonvirtual bus, and In4 outputs an int16.

mdl = 'ex_dataset_for_inports';
open_system(mdl)
Create a Dataset object for the root-level Inport blocks.

ds = createInputDataset(mdl)

ds =

Simulink.SimulationData.Dataset '' with 4 elements

<table>
<thead>
<tr>
<th>Name</th>
<th>BlockPath</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>In1 ''</td>
</tr>
<tr>
<td>2</td>
<td>In2 ''</td>
</tr>
<tr>
<td>3</td>
<td>In3 ''</td>
</tr>
<tr>
<td>4</td>
<td>In4 ''</td>
</tr>
</tbody>
</table>

- Use braces {} to access, modify, or add elements using index.

Replace the placeholder value for the first signal in the Dataset with actual signal values that you want to load into the model.

ds{1} = ds{1}.delsample('Index',[1,2]);
ds{1} = ds{1}.addsample('time',[1 3 3 10],'data',[1 1 5 5]);

Examine the In2 signal.
ds{2}

ans =

struct with fields:
    a: [1x1 timeseries]
    b: [1x1 timeseries]

For In2, create data for bus elements a and b.

ds{2}.a = ds{2}.a.delsample('Index',[1,2]);
ds{2}.a = addsample(ds{2}.a,'time',[1:10],'data',[1:10]);
ds{2}.b = timeseries((1:10)',0.1:.1:1,'Name','sig2_b');

For In3, specify data for element a of the bus, and use ground values for element b.

ds{3}.a = timeseries((1:10)',0.1:.1:1,'Name','sig3_a');

Plot ds.

plot(ds)

Set the Input configuration parameter to ds. Alternatively, you can use the Root Inport Mapper tool to set the Input parameter.

set_param(mdl,'LoadExternalInput','on');
set_param(mdl,'ExternalInput','ds');

Run the simulation. The Inport blocks use the signal data specified in ds or ground values for elements that do not have specified signal data.

sim(mdl)

**Input Arguments**

- **model** — Model for which to generate dataset for root-level Inport blocks or bus element ports
  - character vector | model handle

  Model for which to generate a dataset with an element for each root-level Inport block, specified as a character vector or model handle.
Output Arguments

inports_dataset — Dataset with an element for each root-level Inport block

A Simulink.SimulationData.Dataset object

Dataset with an element for each root-level Inport block, returned as a Simulink.SimulationData.Dataset object.

Related Links

Simulink.SimulationData.Dataset MATLAB timeseries timetable “Create a Dataset Object for Inport Blocks”

Introduced in R2017a
getHardwareImplementation

Class: coder.BuildConfig
Package: coder

Get handle of copy of hardware implementation object

Syntax

hw = bldcfg.getHardwareImplementation()

Description

hw = bldcfg.getHardwareImplementation() returns the handle of a copy of the hardware implementation object.

Input Arguments

bldcfg

coder.BuildConfig object.

Output Arguments

hw

Handle of copy of hardware implementation object.

See Also
getStdLibInfo

Class: coder.BuildConfig
Package: coder

Get standard library information

Syntax

\[\text{[linkLibPath,linkLibExt,execLibExt,libPrefix]} = \]
\text{bldcfg.getStdLibInfo()}\]

Description

\[\text{[linkLibPath,linkLibExt,execLibExt,libPrefix]} = \]
\text{bldcfg.getStdLibInfo()} \text{ returns character vectors representing the:}\n
\begin{itemize}
  \item Standard MATLAB architecture-specific library path
  \item Platform-specific library file extension for use at link time
  \item Platform-specific library file extension for use at run time
  \item Standard architecture-specific library name prefix
\end{itemize}

Input Arguments

bldcfg

coder.BuildConfig object.

Output Arguments

linkLibPath

Standard MATLAB architecture-specific library path specified as a character vector. The character vector can be empty.
linkLibExt

Platform-specific library file extension for use at link time, specified as a character vector. The value is one of '.lib', '.dylib', '.so', ''.

execLibExt

Platform-specific library file extension for use at run time, specified as a character vector. The value is one of '.dll', '.dylib', '.so', ''.

linkPrefix

Standard architecture-specific library name prefix, specified as a character vector. The character vector can be empty.
**getTargetLang**

**Class:** coder.BuildConfig  
**Package:** coder

Get target code generation language

**Syntax**

```python
lang = bldcfg.getTargetLang()
```

**Description**

`lang = bldcfg.getTargetLang()` returns a character vector containing the target code generation language.

**Input Arguments**

- **bldcfg**  
  coder.BuildConfig object.

**Output Arguments**

- **lang**  
  A character vector containing the target code generation language. The value is 'C' or 'C++'.

---

2-205
getToolchainInfo

Class: coder.BuildConfig
Package: coder

Returns handle of copy of toolchain information object

Syntax

tc = bldcfg.getToolchainInfo()

Description

tc = bldcfg.getToolchainInfo() returns a handle of a copy of the toolchain information object.

Input Arguments

bldcfg
coder.BuildConfig object.

Output Arguments

tc
Handle of copy of toolchain information object.

See Also
isCodeGenTarget

**Class:** coder.BuildConfig  
**Package:** coder

Determine if build configuration represents specified target

**Syntax**

\[ tf = \text{bldcfg}.\text{isCodeGenTarget}(\text{target}) \]

**Description**

\[ tf = \text{bldcfg}.\text{isCodeGenTarget}(\text{target}) \] returns true (1) if the code generation target of the current build configuration represents the code generation target specified by `target`. Otherwise, it returns false (0).

**Input Arguments**

**bldcfg**

coder.BuildConfig object.

**target**

Code generation target specified as a character vector or cell array of character vectors.

<table>
<thead>
<tr>
<th>Specify</th>
<th>For code generation target</th>
</tr>
</thead>
<tbody>
<tr>
<td>'rtw'</td>
<td>C/C++ dynamic Library, C/C++ static library, or C/C++ executable</td>
</tr>
<tr>
<td>'sfun'</td>
<td>S-function (Simulation)</td>
</tr>
<tr>
<td>'mex'</td>
<td>MEX-function</td>
</tr>
</tbody>
</table>

Specify `target` as a cell array of character vectors to test if the code generation target of the build configuration represents one of the targets specified in the cell array.
For example:

```matlab
... mytarget = {'sfun','mex'};
tf = bldcfg.isCodeGenTarget(mytarget);
...```

tests whether the build context represents an S-function target or a MEX-function target.

**Output Arguments**

tf

The value is true (1) if the code generation target of the build configuration represents the code generation target specified by `target`. Otherwise, the value is false (0).

**See Also**
coder.target
isMatlabHostTarget

Class: coder.BuildConfig
Package: coder

Determine if hardware implementation object target is MATLAB host computer

Syntax

\[
tf = \text{bldcfg}.\text{isMatlabHostTarget}()
\]

Description

\[
tf = \text{bldcfg}.\text{isMatlabHostTarget}() \text{ returns true (1) if the current hardware implementation object targets the MATLAB host computer. Otherwise, it returns false (0).}
\]

Input Arguments

bldcfg

coder.BuildConfig object.

Output Arguments

tf

Value is true (1) if the current hardware implementation object targets the MATLAB host computer. Otherwise, the value is false (0).

See Also

2-209
coder.ExternalDependency.getDescriptiveName

Class: coder.ExternalDependency
Package: coder

Return descriptive name for external dependency

Syntax

extname = coder.ExternalDependency.getDescriptiveName(bldcfg)

Description

extname = coder.ExternalDependency.getDescriptiveName(bldcfg) returns the name that you want to associate with an “external dependency” on page 2-211. The code generator uses the external dependency name for error messages.

Input Arguments

bldcfg
coder.BuildConfig object. Use coder.BuildConfig methods to get information about the “build context” on page 2-211

You can use this information when you want to return different names based on the build context.

Output Arguments

extname
External dependency name returned as a character vector.
Examples

Return external dependency name

Define a method that always returns the same name.

```matlab
function myextname = getDescriptiveName(~)
    myextname = 'MyLibrary'
end
```

Return external library name based on the code generation target

Define a method that uses the build context to determine the name.

```matlab
function myextname = getDescriptiveName(context)
    if context.isMatlabHostTarget()
        myextname = 'MyLibrary_MatlabHost';
    else
        myextname = 'MyLibrary_Local';
    end
end
```

More About

external dependency

External code interface represented by a class derived from a `coder.ExternalDependency` class. The external code can be a library, object files, or C/C++ source.

build context

Information used by the build process including:

- Target language
- Code generation target
• Target hardware
• Build toolchain

See Also
coder.BuildConfig | coder.ExternalDependency | coder.ceval |
encoder.updateBuildInfo

Topics
“Develop Interface for External C/C++ Code” (MATLAB Coder)
“Build Process Customization” (MATLAB Coder)
“Integrate External/Custom Code” (MATLAB Coder)
**coder.ExternalDependency.isSupportedContext**

**Class:** coder.ExternalDependency  
**Package:** coder

Determine if build context supports external dependency

**Syntax**

```matlab
tf = coder.ExternalDependency.isSupportedContext(bldcfg)
```

**Description**

`tf = coder.ExternalDependency.isSupportedContext(bldcfg)` returns true (1) if you can use the “external dependency” on page 2-214 in the current “build context” on page 2-214. You must provide this method in the class definition for a class that derives from `coder.ExternalDependency`.

If you cannot use the “external dependency” on page 2-214 in the current “build context” on page 2-214, display an error message and stop code generation. The error message must describe why you cannot use the external dependency in this build context. If the method returns false (0), the code generator uses a default error message. The default error message uses the name returned by the `getDescriptiveName` method of the `coder.ExternalDependency` class.

Use `coder.BuildConfig` methods to determine if you can use the external dependency in the current build context.

**Input Arguments**

`bldcfg`  
`coder.BuildConfig` object. Use `coder.BuildConfig` methods to get information about the “build context” on page 2-214.
Output Arguments

tf

Value is true (1) if the build context supports the external dependency.

Examples

Report error when build context does not support external library

This method returns true(1) if the code generation target is a MATLAB host target. Otherwise, the method reports an error and stops code generation.

Write isSupportedContext method.

```matlab
function tf = isSupportedContext(ctx)
    if ctx.isMatlabHostTarget()
        tf = true;
    else
        error('adder library not available for this target');
    end
end
```

More About

external dependency

External code interface represented by a class derived from coder.ExternalDependency class. The external code can be a library, object file, or C/C++ source.

build context

Information used by the build process including:

• Target language
• Code generation target
• Target hardware
• Build toolchain

See Also

coder.BuildConfig | coder.ExternalDependency | coder.ceval |
coder.updateBuildInfo

Topics

“Develop Interface for External C/C++ Code” (MATLAB Coder)
“Build Process Customization” (MATLAB Coder)
“Integrate External/Custom Code” (MATLAB Coder)
coder.ExternalDependency.updateBuildInfo

**Class:** coder.ExternalDependency  
**Package:** coder

Update build information

**Syntax**

coder.ExternalDependency.updateBuildInfo(buildInfo, bldcfg)

**Description**

coder.ExternalDependency.updateBuildInfo(buildInfo, bldcfg) updates the build information object whose handle is buildInfo. After code generation, the build information object has standard information. Use this method to provide additional information required to link to external code. Use coder.BuildConfig methods to get information about the “build context” on page 2-217.

You must implement this method in a subclass of coder.ExternalDependency. For an example, see coder.ExternalDependency.

**Input Arguments**

**buildInfo**

Handle of build information object.

**bldcfg**

coder.BuildConfig object. Use coder.BuildConfig methods to get information about the “build context” on page 2-217.
Limitations

• The build information method `AddIncludeFiles` has no effect in a `coder.ExternalDependency.updateBuildInfo` method.

More About

build context

Information used by the build process including:

• Target language
• Code generation target
• Target hardware
• Build toolchain

See Also

coder.BuildConfig | coder.ExternalDependency | coder.ceval | coder.updateBuildInfo

Topics

“Develop Interface for External C/C++ Code“ (MATLAB Coder)
“Build Process Customization“ (MATLAB Coder)
“Integrate External/Custom Code“ (MATLAB Coder)
convertToSLDataset

Convert contents of MAT-file to Simulink.SimulationData.Dataset object

Syntax

success=convertToSLDataset(source,destination)
success=convertToSLDataset(source,destination,datasetname)

Description

success=convertToSLDataset(source,destination) converts the contents of a MAT-file (source) to a destination MAT-file (destination).

success=convertToSLDataset(source,destination,datasetname) names the dataset datasetname.

When converting structure signal data, the function names the signal using the value contained in the label field of the structure signal field, such as: mySignal.signal(1).label.

This function ignores time expressions in source.

Examples

Save Signals to Dataset in file2.mat

Save signals from file1.mat to a dataset named file1 in file2.mat.

success=convertToSLDataset('file1.mat','file2.mat')

Save Signals to Dataset Named myDataset in file2.mat

Save signals from file1.mat to a dataset named myDataset in file2.mat.
success=convertToSLDataset('file1.mat','file2.mat','myDataset')

Input Arguments

source — MAT-file
character vector
MAT-file that contains Simulink inputs.

destination — MAT-file
character vector
MAT-file to contain Simulink.SimulationData.Dataset converted from contents of source.

datasetname — Data set name
character vector
Data set name for new Simulink.SimulationData.Dataset object.

Output Arguments

success — Outcome of conversion
binary
Outcome of conversion, specified as binary:

- 1
  Conversion is successful.
- 0
  Conversion is not successful.

See Also

Introduced in R2016a
**copy**

Copy a configuration set

**Syntax**

```matlab
copyCs = copy(cs)
```

**Description**

`copyCs = copy(cs)` returns a copy of a configuration set.

**Examples**

**Attach New Configuration Set to a Model**

Create a copy of a configuration set and attach it to a model.

Get the active configuration set for your model.

```matlab
activeConfig = getActiveConfigSet ('vdp');
```

Copy the active configuration set.

```matlab
develConfig = copy(activeConfig);
```

Give the copied configuration set a name.

```matlab
develConfig.Name = 'develConfig';
```

Attach the new configuration set to the model.
attachConfigSet('vdp',develConfig);

**Input Arguments**

*cs — Configuration set*  
`ConfigSet object`  
Configuration set object to copy, specified as a `ConfigSet` object.

**Output Arguments**

*copyCs — Copy of configuration set*  
`ConfigSet object`  
A copy of the configuration set, specified as a `ConfigSet` object.

**See Also**

**Topics**

“About Configuration Sets”  
“Manage a Configuration Set”

*Introduced before R2006a*
createCategory

(Not recommended) Create category of project labels

**Note** `simulinkproject` is not recommended. Use `currentProject` or `openProject` instead. For more information, see “Compatibility Considerations”.

**Syntax**

createCategory(proj, categoryName)
createCategory(proj, categoryName, dataType)
createCategory(proj, categoryName, dataType, single-valued)

**Description**

`createCategory(proj, categoryName)` creates a new category of labels `categoryName` in the project `proj`.

`createCategory(proj, categoryName, dataType)` specifies the class of data to store in labels of the new category.

`createCategory(proj, categoryName, dataType, single-valued)` specifies a single-valued category, where you can attach only one label from the category to a file. If you do not specify single-valued, then you can attach multiple labels from the category to a file.

**Examples**

**Create a New Category of Labels for File Ownership**

Create a new category of labels for file ownership, and attach a new label and label data to a file.

Open the airframe project and create a project object.
Create a new category of labels, called **Engineers**, to denote file ownership in a project. These labels have the char datatype for attaching character vector data.

```matlab
createCategory(proj,'Engineers','char');
```

Use **findCategory** to get the new category.

```matlab
enGINEERSCATEGORY = findCategory(proj,'Engineers');
```

Create labels in the new category.

```matlab
createLabel(EENGINEERSCATEGORY,'Tom');
createLabel(EENGINEERSCATEGORY,'Dick')
createLabel(EENGINEERSCATEGORY,'Harry')
```

Attach one of the new labels to a file in the project.

```matlab
myfile = findFile(proj,'models/AnalogControl.slx')
addLabel(myfile,'Engineers','Tom');
```

Get the label and add data.

```matlab
label = findLabel(file,'Engineers','Tom');
label.Data = 'Maintenance responsibility';
disp(label)
```

Label with properties:

- File: [1x80 char]
- Data: 'Maintenance responsibility'
- DataType: 'char'
- Name: 'Tom'
- CategoryName: 'Engineers'

### Create a New Category of Labels with Datatype Double

Open the airframe project and create a project object.

```matlab
sldemo_slproject_airframe
proj = simulinkproject;
```
Create a new category of labels.

```matlab
createCategory(proj,'Coverage','double')
```

category =

Category with properties:

- Name: 'Coverage'
- DataType: 'double'
- LabelDefinitions: []

Find out what you can do with the new category.

```matlab
category = findCategory(proj, 'Coverage');
methods(category)
```

Methods for class slproject.Category:

- findLabel
- removeLabel
- createLabel

**Create a Single-Valued Category**

Open the airframe project and create a project object.

```matlab
sldemo_slproject_airframe
proj = simulinkproject;
```

Create a category of labels for file ownership, and specify single-valued to restrict only one label in the category per file.

```matlab
createCategory(proj,'Engineers','char', 'single-valued');
```

**Input Arguments**

- **proj** — Project
  
  Project, specified as a project object. Use simulinkproject to create a project object to manipulate a project at the command line.
**categoryName — Name of category**
character vector

Name of the category of labels to create, specified as a character vector.

**dataType — Class of data to store in labels**
character vector

The class of data to store in labels in the new category, specified as a character vector.

**single-valued — Single-valued category**
character vector

Single-valued category, specified as a character vector. Single-valued means you can attach only one label from the category to a file. If you do not specify single-valued, then you can attach multiple labels from the category to a file.

**Tips**

After you create a new category, you can create labels in the new category. See `createLabel`.

**Compatibility Considerations**

**Simulink project API is not recommended**

*Not recommended starting in R2019a*

Starting in R2019a, instead of `simulinkproject` and related functions, use the `currentProject` or `openProject` functions in MATLAB. The Simulink project API will continue to be supported, but, after R2019a, new features might be available only if you use the new MATLAB project API. There are no plans to remove the Simulink project API at this time.

The new MATLAB project API is part of the R2019a functionality enabling you to use projects in MATLAB, with or without Simulink. You can now share projects with users who do not have Simulink.
See Also

Functions
createLabel | simulinkproject

Introduced in R2013a
createLabel

(Not recommended) Define project label

**Note** `simulinkproject` is not recommended. Use `currentProject` or `openProject` instead. For more information, see “Compatibility Considerations”.

**Syntax**

createLabel(category,newLabelName)

**Description**

`createLabel(category,newLabelName)` creates a new label, `newLabelName`, in a category. Use this syntax if you previously got a category by accessing a `Categories` property, e.g., using a command like `proj.Categories(1)`.

Use `addLabel` instead to create and attach a new label in an existing category using a single step.

Use `createCategory` first if you want to make a new category of labels.

**Examples**

**Create a New Label**

Open the airframe project and create a project object.

```matlab
sldemo_slproject_airframe
proj = simulinkproject;
```

Examine the first existing category.

```matlab
cat = proj.Categories(1)
```
Define a new label in the category.

createLabel(cat,'Future');

Create a New Category of Labels for File Ownership

Open the airframe project and create a project object.

sldemo_slproject_airframe
proj = simulinkproject;

Create creates a new category of labels called Engineers which can be used to denote file ownership in a project. These labels have the char datatype for attaching character vector data.

createCategory(proj,'Engineers','char');

Use findCategory to get the new category.

engineersCategory = findCategory(proj,'Engineers');

Create labels in the new category.

createLabel(engineersCategory,'Tom');
createLabel(engineersCategory,'Dick');
createLabel(engineersCategory,'Harry');

Attach one of the new labels to a file in the project.

myfile = findFile(proj,'models/AnalogControl.slx')
addLabel(myfile,'Engineers', 'Tom');

Get the label and add data.
label = findLabel(myfile,'Engineers','Tom');
label.Data = 'Maintenance responsibility';
disp(label)

Label with properties:

File: [1x80 char]
Data: 'Maintenance responsibility'
DataType: 'char'
Name: 'Tom'
CategoryName: 'Engineers'

Input Arguments

category — Category

category object

Category for the new label, specified as a category object. Get the category by accessing a Categories property, e.g. with a command like proj.Categories(1), or use findCategory. To create a new category, use createCategory.

newLabelName — The name of the new label to define

character vector

The name of the new label to define, specified as a character vector.

Compatibility Considerations

Simulink project API is not recommended

Not recommended starting in R2019a

Starting in R2019a, instead of simulinkproject and related functions, use the currentProject or openProject functions in MATLAB. The Simulink project API will continue to be supported, but, after R2019a, new features might be available only if you use the new MATLAB project API. There are no plans to remove the Simulink project API at this time.
The new MATLAB project API is part of the R2019a functionality enabling you to use projects in MATLAB, with or without Simulink. You can now share projects with users who do not have Simulink.

See Also

Functions
addLabel | createCategory

Introduced in R2013a
addStartupFile

(Not recommended) Add startup file to project

**Note** simulinkproject is not recommended. Use currentProject or openProject instead. For more information, see “Compatibility Considerations”.

**Syntax**

addStartupFile(proj, file)

**Description**

addStartupFile(proj, file) adds a startup file to the project proj. Startup files automatically run (.m and .p files), load (.mat files), or open (Simulink models) when you open the project.

**Examples**

**Add a Startup File**

Open the airframe project and create a project object.

```matlab
sldemo_slproject_airframe
proj = simulinkproject;
```

Create a new file.

```matlab
filepath = fullfile(proj.RootFolder, 'new_model.slx')
    new_system('new_model');
    save_system('new_model', filepath)
```

Add the new model to the project.

```matlab
projectFile = addFile(proj, filepath)
```
Automatically open the model when the project opens, by making it a startup file.

addStartupFile(proj, filepath);

**Input Arguments**

proj — Project

*project object*

Project, specified as a project object. Use `simulinkproject` to create a project object to manipulate a project at the command line.

file — Path of file

*character vector*

Path of the file to add relative to the project root folder, including the file extension, specified as a character vector. The file must be within the root folder.

Example: ‘models/myModelName.slx’

**Compatibility Considerations**

**Simulink project API is not recommended**

*Not recommended starting in R2019a*

Starting in R2019a, instead of `simulinkproject` and related functions, use the `currentUserProject` or `openProject` functions in MATLAB. The Simulink project API will continue to be supported, but, after R2019a, new features might be available only if you use the new MATLAB project API. There are no plans to remove the Simulink project API at this time.

The new MATLAB project API is part of the R2019a functionality enabling you to use projects in MATLAB, with or without Simulink. You can now share projects with users who do not have Simulink.

**See Also**

`simulinkproject`
Introduced in R2017b
addShutdownFile

(Not recommended) Add shutdown file to project

**Note** simulinkproject is not recommended. Use currentProject or openProject instead. For more information, see .

**Syntax**

addShutdownFile(proj, file)

**Description**

addShutdownFile(proj, file) adds a shutdown file to the project proj. When you close the project, it runs the shutdown file automatically. Use the shutdown list to specify executable MATLAB code to run as the project shuts down.

**Examples**

**Add a Shutdown File**

Open the airframe project and create a project object.

```matlab
sldemo_slproject_airframe
proj = simulinkproject;
```

Specify executable MATLAB code to run as the project shuts down.

```matlab
filepath = fullfile('utilities', 'rebuild_s_functions.m');
```

Automatically run the file when the project closes, by making it a shutdown file.
addShutdownFile(project, filepath);

**Input Arguments**

`proj — Project`  
project object

Project, specified as a project object. Use `simulinkproject` to create a project object to manipulate a project at the command line.

`file — Path of file`  
character vector

Path of the MATLAB file to add relative to the project root folder, including the file extension, specified as a character vector. The file must be within the root folder.

Example: `'utilities/myscript.m'`

**Compatibility Considerations**

**Simulink project API is not recommended**

*Not recommended starting in R2019a*

Starting in R2019a, instead of `simulinkproject` and related functions, use the `currentProject` or `openProject` functions in MATLAB. The Simulink project API will continue to be supported, but, after R2019a, new features might be available only if you use the new MATLAB project API. There are no plans to remove the Simulink project API at this time.

The new MATLAB project API is part of the R2019a functionality enabling you to use projects in MATLAB, with or without Simulink. You can now share projects with users who do not have Simulink.

**See Also**

`simulinkproject`
Introduced in R2017b
removeStartupFile

(Not recommended) Remove startup file from project startup list

**Note** simulinkproject is not recommended. Use currentProject or openProject instead. For more information, see “Compatibility Considerations”.

**Syntax**

removeStartupFile(proj, file)

**Description**

removeStartupFile(proj, file) removes the startup file from the startup list in the project proj.

**Examples**

**Remove a Startup File**

Open the airframe project and create a project object.

```matlab
sldemo_slproject_airframe
proj = simulinkproject;
```

Create a new file.

```matlab
filepath = fullfile(proj.RootFolder, 'new_model.slx')
    new_system('new_model');
    save_system('new_model', filepath)
```

Add the new model to the project.

```matlab
projectFile = addFile(proj, filepath)
```
Automatically open the model when the project opens, by making it a startup file.

```
addStartupFile(proj, filepath);
```

Remove the startup file.

```
removeStartupFile(proj, filepath);
```

## Input Arguments

**proj** — Project

Project object

- Project, specified as a project object. Use `simulinkproject` to create a project object to manipulate a project at the command line.

**file** — Path of file

Character vector

- Path of the file to add relative to the project root folder, including the file extension, specified as a character vector. The file must be within the root folder.

Example: ‘models/myModelName.slx’

## Compatibility Considerations

**Simulink project API is not recommended**

*Not recommended starting in R2019a*

Starting in R2019a, instead of `simulinkproject` and related functions, use the `currentProject` or `openProject` functions in MATLAB. The Simulink project API will continue to be supported, but, after R2019a, new features might be available only if you use the new MATLAB project API. There are no plans to remove the Simulink project API at this time.

The new MATLAB project API is part of the R2019a functionality enabling you to use projects in MATLAB, with or without Simulink. You can now share projects with users who do not have Simulink.
See Also

simulinkproject

Introduced in R2017b
removeShutdownFile

(Not recommended) Remove shutdown file from project shutdown list

**Note** `simulinkproject` is not recommended. Use `currentProject` or `openProject` instead. For more information, see “Compatibility Considerations”.

**Syntax**

```matlab
removeShutdownFile(proj, file)
```

**Description**

`removeShutdownFile(proj, file)` removes the shutdown file from the shutdown list in the project `proj`.

**Examples**

**Remove a Shutdown File**

Open the airframe project and create a project object.

```matlab
sldemo_slproject_airframe
proj = simulinkproject;
```

Specify executable MATLAB code to run as the project shuts down.

```matlab
filepath = fullfile('utilities', 'rebuild_s_functions.m');
```

Automatically run the file when the project closes, by making it a shutdown file.

```matlab
addShutdownFile(project, filepath);
```

Remove the shutdown file.
removeShutdownFile(project, filepath);

**Input Arguments**

**proj — Project**
project object

Project, specified as a project object. Use `simulinkproject` to create a project object to manipulate a project at the command line.

**file — Path of file**
character vector

Path of the MATLAB file to add relative to the project root folder, including the file extension, specified as a character vector. The file must be within the root folder.

Example: ‘utilities/myscript.m’

**Compatibility Considerations**

**Simulink project API is not recommended**

*Not recommended starting in R2019a*

Starting in R2019a, instead of `simulinkproject` and related functions, use the `currentProject` or `openProject` functions in MATLAB. The Simulink project API will continue to be supported, but, after R2019a, new features might be available only if you use the new MATLAB project API. There are no plans to remove the Simulink project API at this time.

The new MATLAB project API is part of the R2019a functionality enabling you to use projects in MATLAB, with or without Simulink. You can now share projects with users who do not have Simulink.

**See Also**

`simulinkproject`
Introduced in R2017b
**delete_block**

Delete blocks from Simulink system

**Syntax**

delete_block(blockArg)

**Description**

delete_block(blockArg) deletes the specified blocks from a system. Open the system before you delete blocks.

**Examples**

**Delete Block Using Full Path Name**

Delete the block Mu from the vdp system.

```matlab
open_system('vdp')
delete_block('vdp/Mu')
```

**Delete Block Using Block Handle**

Delete the block Out2 from the vdp system using the block handle.

Open the vdp system.

```matlab
open_system('vdp')
```

Interactively select the block Out1. Get the block’s handle and assign it to the variable `Out1_handle`. Delete the block using the handle.
Out1_handle = get_param(gcb,'Handle');
delete_block(Out1_handle)

Delete Blocks Using Vector of Handles

Delete three blocks from the vdp system.

Open the vdp system. Add three blocks and assign their handles to variables.

open_system('vdp')
Constant_handle = add_block('built-in/Constant','vdp/MyConstant');
Gain_handle = add_block('built-in/Gain','vdp/MyGain');
Outport_handle = add_block('built-in/Outport','vdp/MyOutport');

Delete the blocks you added using a vector of handles.

delete_block([Constant_handle Gain_handle Outport_handle])

Input Arguments

blockArg — Blocks to delete
full path name | handle | vector of handles | 1-D cell array of handles or block path names

Blocks to delete, specified as the full block path name, a handle, a vector of handles, or a 1-D cell array of handles or block path names.

Example: 'vdp/Mu'
Example: [handle1 handle2]
Example: {'vdp/Mu' 'vdp/Out1' 'vdp/Out2'}

See Also

add_block

Introduced before R2006a
delete_line

Delete line from Simulink model

Syntax

delem_line(sys,out,in)
delem_line(sys,point)
delem_line(lineHandle)

Description

delem_line(sys,out,in) deletes the line from the model or subsystem sys that connects the output port out to the input port in.

delem_line(sys,point) deletes the line that includes the point point.

delem_line(lineHandle) deletes the line using the line handle.

Examples

Remove Line Using Block Port Names

For the model vdp, remove the line connecting the Product block with the Gain block.

load_system('vdp');
delem_line('vdp','Product/1','Mu/1');

Remove Line Using Line Handle

For the model vdp, remove a line using the line handle. You can get the line handle using different techniques.
load_system('vdp');
h = get_param('vdp/Mu','LineHandles');
delete_line(h.Outport(1));

Get a line handle when you create the line. Delete the line using that handle.

a = add_line('vdp','Mu/1','Sum/2');
delete_line(a)

**Delete a Line Using a Point**

You can use a point on the line to delete the whole line.

Find the port coordinates for the block Mu in the model vdp.

```matlab
open_system('vdp');
mu = get_param('vdp/Mu','PortConnectivity');
mu.Position
ans = 1×2
    190   150
ans = 1×2
    225   150
```

The line that connects the Mu block to the Sum block starts at the output port, which is at (225,150). You can use any point to the right of that point along the same x-axis to delete the line.

```matlab
delete_line('vdp',[230,150]);
```

**Delete Segments of Branched Lines**

Use `delete_line` with branched lines to remove the segment for any connection.

Open the model vdp.

```matlab
open_system('vdp');
```
Delete the line from x1 to the Out1 block. This command deletes only the segment of the line that connects the branch to the specified block.

```
delete_line('vdp','x1/1','Out1/1')
```

Delete the line segment from x2 to the Mux.

```
delete_line('vdp','x2/1','Mux/2')
```

Delete the line segment from x2 to the Product block.

```
delete_line('vdp','x2/1','Product/2')
```
Input Arguments

sys — Model or subsystem to delete line from
character vector

Model or subsystem to delete the line from, specified as a character vector.
Example: 'vdp', 'f14/Controller'

out — Block output port to delete line from
block/port name or number character vector | port handle

Block output port to delete line from, specified as either:

- The block name, a slash, and the port name or number. Most block ports are numbered from top to bottom or from left to right. For a state port, use the port name State instead of a port number.
- The port handle that you want to delete the line from.

Use 'PortHandles' with get_param to get the handles.
Example: 'Mu/1', 'Subsystem/2'

in — Block input port to delete line from
block/port name or number character vector | port handle
Block input port to delete line from, specified as either:

- The block name, a slash, and the port name or number. The port name on:
  - An enabled subsystem is Enable.
  - A triggered subsystem is Trigger.
  - If Action and Switch Case Action subsystems is Action.
- The port handle that you want to delete the line from.

Use 'PortHandles' with get_param to get handles.

Example: 'Mu/1', 'Subsystem/2'

**point** — Point on the line you want to delete

1-by-2 matrix

Point that falls on the line you want to delete, specified as a 1-by-2 matrix.

Example: [150 200]

**lineHandle** — Handle of the line you want to delete

Handle of the line you want to delete. You can get the line handle by using get_param with the 'LineHandles' option or by assigning the line to a handle when you create it programmatically.

See Also

add_line | get_param

Introduced before R2006a
**delete_param**

Delete system parameter added via `add_param` command

**Syntax**

```
delete_param('sys','parameter1','parameter2',...)```

**Description**

This command deletes parameters that were added to the system using the `add_param` command. The command displays an error message if a specified parameter was not added with the `add_param` command.

**Examples**

The following example

```
add_param('vdp','DemoName','VanDerPolEquation','EquationOrder','2')
delete_param('vdp','DemoName')
```

adds the parameters DemoName and EquationOrder to the vdp system, then deletes DemoName from the system.

**See Also**

`add_param`

*Introduced before R2006a*
dependencies.fileDependencyAnalysis

Find model file dependencies

Syntax

files = dependencies.fileDependencyAnalysis('modelname')
[files, missing] = dependencies.fileDependencyAnalysis('modelname')
[files, missing, depfile] = dependencies.fileDependencyAnalysis('modelname')
[files, missing, depfile, manifestfile] = dependencies.fileDependencyAnalysis('modelname', 'manifestfile')

Description

files = dependencies.fileDependencyAnalysis('modelname') returns files, a cell array of character vectors containing the full paths of all existing files referenced by the model modelname.

[files, missing] = dependencies.fileDependencyAnalysis('modelname') returns files, all existing files referenced by the model modelname, and any referenced files that cannot be found in missing.

[files, missing, depfile] = dependencies.fileDependencyAnalysis('modelname') also returns depfile, the full path of the user dependencies (.smd) file, if it exists, that stores the names of any files you manually added or excluded.

[files, missing, depfile, manifestfile] = dependencies.fileDependencyAnalysis('modelname', 'manifestfile') also creates a manifest file with the name and path specified in manifestfile.

Input Arguments

modelname

Character vector specifying the name of the model to analyze for dependencies.
manifestfile

(Optional) Character vector to specify the name of the manifest file to create. You can specify a full path or just a file name (in which case the file is created in the current folder). The function adds the suffix .smf to the user-specified name.

Output Arguments

files

A cell array of character vectors containing the full-paths of all existing files referenced by the model modelname. If there is only one dependency, the return is a character vector. If there are no dependencies, the return is empty.

Default: []

missing

A cell array of character vectors containing the names of any files that are referenced by the model modelname, but cannot be found.

Default: []

depfile

Character vector containing the full path of a user dependencies (.smd) file, if it exists, that stores the names of any files you manually added or excluded. Simulink uses the .smd file to remember your changes the next time you generate a manifest. See “Edit Manifests”.

Default: []

manifestfile

Character vector containing the name and path of the new manifest file.

Default: []
Examples

The following code analyses the model `mymodel` for file dependencies:

```matlab
files = dependencies.fileDependencyAnalysis('mymodel')
```

If you try dependency analysis on an example model, it returns an empty list of required files because the standard MathWorks installation includes all the files required for the example models.

Tip

If you try dependency analysis on an example model, it returns an empty list of required files because the standard MathWorks installation includes all the files required for the example models.

Alternatives

If your file is in a project, use `listRequiredFiles` instead.

You can interactively run dependency analysis in a project. See “Run Dependency Analysis”.

To create a report to identify where dependencies arise, find required toolboxes, and for more control over dependency analysis options, you can interactively generate a manifest and report. See “Analyze Model Dependencies”.

To programmatically check which `toolboxes` are required, see `dependencies.toolboxDependencyAnalysis`.

See Also

`dependencies.toolboxDependencyAnalysis` | `listRequiredFiles`

Topics

“What Are Model Dependencies?”
Introduced in R2012a
dependencies.toolboxDependencyAnalysis

Find toolbox dependencies

Syntax

names = dependencies.toolboxDependencyAnalysis(files_in)

[ names, folders ] = dependencies.toolboxDependencyAnalysis(files_in)

Description

names = dependencies.toolboxDependencyAnalysis(files_in) returns names, a cell array of toolbox names required by the files in files_in.

[ names, folders ] = dependencies.toolboxDependencyAnalysis(files_in) returns toolbox names and also a cell array of the toolbox folders.

Tip In a project, you can interactively run dependency analysis. You can find the required toolboxes for the whole project or for selected files. You can see which products a new team member requires to use the project, or find which file is introducing a product dependency. See “Find Required Toolboxes”.

Input Arguments

files_in

Cell array of character vectors containing .m, .mdl, or .slx files on the MATLAB path. Simulink model names (without file extension) are also allowed.

Default: [ ]
Output Arguments

names
Cell array of toolbox names required by the files in files_in.

folders
(Optional) Cell-array of the required toolbox folders.

Examples

The following code reports the detectable required toolboxes for the model vdp:

```matlab
files_in={'vdp'};
names = dependencies.toolboxDependencyAnalysis(files_in)
names =
    'MATLAB'    'Simulink'    'Simulink Coder'
```

To find all detectable toolbox dependencies of your model and the files it depends on:

1 Call fileDependencyAnalysis on your model.
   
   For example:
   ```matlab
   files = dependencies.fileDependencyAnalysis('mymodel')
   
   files =
       'C:\Work\foo.m'
       'C:\Work\mymodel.mdl'
   ```

2 Call toolboxDependencyAnalysis on the files output of step 1.
   
   For example:
   ```matlab
   tbxes = dependencies.toolboxDependencyAnalysis(files)
   
   tbxes =
   [1x24 char]    'MATLAB'    'Simulink Coder'    'Simulink'
   ```

To view long product names examine the tbxes cell array as follows:
The function `dependencies.toolboxDependencyAnalysis` looks for toolbox dependencies of the files in `files_in` but does not analyze any subsequent dependencies. See “Examples” on page 2-256.

For command-line dependency analysis, the analysis uses the default settings for analysis scope to determine required toolboxes. For example, if you have code generation products, then the check **Find files required for code generation** is on by default and Simulink Coder is always reported as required. See “Required Toolboxes” in the manifest documentation for more examples of how your installed products and analysis scope settings can affect reported toolbox requirements.

**Alternatives**

In a project, you can interactively run dependency analysis and find the required toolboxes for the whole project or for selected files. See “Find Required Toolboxes”.

For a model that is not in a project, you can interactively generate a manifest and report. You can create a report to identify where dependencies arise, and control dependency analysis options. See “Analyze Model Dependencies”.

To programmatically check which files are required, see `dependencies.fileDependencyAnalysis`.  

```
tbxes{:}
ans = 
  Image Processing Toolbox
ans = 
  MATLAB
ans = 
  Simulink Coder
ans = 
  Simulink
```
See Also
dependencies.fileDependencyAnalysis

Topics
“Dependency Analysis”
“What Is Dependency Analysis?”

Introduced in R2012a
**depview**

Display graph of model referencing dependencies with or without library dependencies

**Syntax**

depview(sys)

depview(sys,Name,Value)

**Description**

depview opens the “Model Dependency Viewer”. While view_mdlrefs also opens the Model Dependency Viewer, depview provides programmatic options that allow you to open a specific configuration of the Model Dependency Viewer.

depview(sys) opens the Model Dependency Viewer, which displays a graph of dependencies for the model or library specified by sys.

depview(sys,Name,Value) opens the Model Dependency Viewer as specified by one or more Name,Value pair arguments.

**Examples**

**Open Model Dependency Viewer with Default Settings**

Open the default Model Dependency Viewer for the model sldemo_mdlref_depgraph.

depview('sldemo_mdlref_depgraph');
In Model Files view, node appearance corresponds to file type.

**Open Model Dependency Viewer with Custom Settings**

Open the Model Dependency Viewer for the model `sldemo_mdlref_depgraph` in **Model Instances** view with a Horizontal layout.

```
depview('sldemo_mdlref_depgraph','ModelReferenceInstance',true,'ShowHorizontal',true);
```
In **Model Instances** view, node appearance corresponds to simulation mode.

**Input Arguments**

```plaintext
sys — Name or path of model or library
    ' ' (default) | character vector
```

Name or path of model or library, specified as a character vector.

Data Types: `char`

**Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example:
```plaintext
depview('sldemo_mdlref_depgraph','ModelReferenceInstance',true,'ShowFullPath',true,'ShowHorizontal',true)
```
**FileDependenciesExcludingLibraries — Graph with only models**
false (default) | true

To open a Model Dependency Viewer that shows only models, set this parameter to 'true'. In this view, node appearance corresponds to file type.

**Dependencies**

You can only set one of 'FileDependenciesExcludingLibraries', 'FileDependenciesIncludingLibraries', and 'ModelReferenceInstance' to 'true'.

**FileDependenciesIncludingLibraries — Graph with models and user-defined libraries**
true (default) | false

To open a Model Dependency Viewer that shows models and user-defined libraries, use the default setting. In this view, node appearance corresponds to file type.

**Dependencies**

To set this parameter to 'false', enable 'FileDependenciesExcludingLibraries' or 'ModelReferenceInstance'. You can only set one of 'FileDependenciesExcludingLibraries', 'FileDependenciesIncludingLibraries', and 'ModelReferenceInstance' to 'true'.

**FactoryDependencies — Graph with models, user-defined libraries, and built-in libraries**
false (default) | true

To open a Model Dependency Viewer that shows models, user-defined libraries, and built-in libraries, set this parameter to 'true'. In this view, node appearance corresponds to file type.

**Dependencies**

To enable this option, set 'FileDependenciesIncludingLibraries' to 'true'.

**ModelReferenceInstance — Graph with only models that shows each model instance separately**
false (default) | true
To open a Model Dependency Viewer that shows each instance of a model as a separate node in the graph, set this parameter to 'true'. In this view, node appearance corresponds to simulation mode.

**Dependencies**

You can only set one of 'FileDependenciesExcludingLibraries', 'FileDependenciesIncludingLibraries', and 'ModelReferenceInstance' to 'true'.

**ShowFullPath — Full path for referenced models and libraries**
false (default) | true

To open a Model Dependency Viewer that shows the full path from the top model to each referenced model or library, set this parameter to 'true'. By default, the Model Dependency Viewer shows only the file name in each node. This parameter is available for only the **Model Instances** view.

**Dependencies**

To enable this option, set 'ModelReferenceInstance' to 'true'.

**ShowHorizontal — Horizontal dependency display**
false (default) | true

To open a Model Dependency Viewer that shows referenced models and libraries to the right of their parents, set this parameter to 'true'. By default, the Model Dependency Viewer shows referenced models and libraries below their parents.

**See Also**
Model | find_mdlrefs | view_mdlrefs

**Topics**
“Model References”
“Model Dependency Viewer”

**Introduced in R2006b**
**detachConfigSet**

Dissociate configuration set or configuration reference from model

**Syntax**

detachConfigSet(model, configObjName)

**Arguments**

model
   The name of an open model, or gcs to specify the current model

configObjName
   The name of a configuration set (Simulink.ConfigSet) or configuration reference (Simulink.ConfigSetRef)

**Description**

detachConfigSet detaches the configuration set or configuration reference (configuration object) specified by configObjName from model. If no such configuration object is attached to the model, an error occurs.

**Examples**

The following example detaches the configuration object named DevConfig from the current model. The code is the same whether DevConfig is a configuration set or configuration reference.

detachConfigSet(gcs, 'DevConfig');
See Also
attachConfigSet | attachConfigSetCopy | closeDialog | getMaxiveConfigSet | getConfigSet | getConfigSets | openDialog | setActiveConfigSet

Topics
“Manage a Configuration Set”
“Manage a Configuration Reference”

Introduced in R2006a
disableimplicitsignalresolution

Convert model to use only explicit signal resolution

Syntax

`retVal = disableimplicitsignalresolution('model')`
`retVal = disableimplicitsignalresolution('model', displayOnly)`

Description

`retVal = disableimplicitsignalresolution('model')` inputs a model, reports all signals and states that implicitly resolve to signal objects, and converts the model to resolve only signals and states that explicitly require it. The report and any changes are limited to the model itself; they do not include blocks that are library links.

Before executing this function, ensure that all relevant Simulink data objects are defined in the base workspace or a data dictionary to which the model is linked. The function ignores any data objects that are defined elsewhere.

The function scans `model`, returns a structure of handles to signals and states that resolve implicitly to signal objects, and performs the following operations on `model`:

- Search the model for all output ports and block states that resolve to Simulink signal objects.
- Modify these ports and blocks to enforce signal object resolution in the future.
- Set the model's `SignalResolutionControl` parameter to 'UseLocalSettings' (GUI: **Explicit Only**).

  If `SignalResolutionControl` is already set to 'UseLocalSettings' or to 'None', the function takes no action and returns a warning.

- If any Stateflow output data resolves to a Simulink signal object:
  - Turn off hierarchical scoping of signal objects from within the Stateflow chart.
  - Explicitly label the output signal of the Stateflow chart.
• Enforce signal object resolution for this signal in the future.

Any changes made by `disableimplicitsignalresolution` permanently change the model. Be sure to back up the model before calling the function with `displayOnly` defaulted to or specified as false.

`retVal = disableimplicitsignalresolution('model', displayOnly)` is equivalent to `disableimplicitsignalresolution(model)` if `displayOnly` is false.

If `displayOnly` is true, the function returns a structure of handles to signals and states that resolve implicitly to signal objects, but leaves the model unchanged.

**Input Arguments**

displayOnly

Boolean specifying whether to change the model (`false`) or just generate a report (`true`).

Default: `false`

model

Model name or handle

**Output Arguments**

retVal

A MATLAB structure containing:

Signals

Handles to ports with signal names that resolve to signal objects

States

Handles to blocks with states that resolve to signal objects

**See Also**

Simulink.Signal
Topics
“Data Validity Diagnostics Overview”
“Symbol Resolution”

Introduced in R2007a
docblock

Get or set editor invoked by Simulink DocBlock

Syntax

docblock(setEditorType,command)
editCommand = docblock(getEditorType)

Description

docblock(setEditorType,command) uses the specified command to set the editor opened by double-clicking a DocBlock block.

By default, a DocBlock block opens Microsoft Word to edit HTML or RTF files. If Word is not available on your system, the block opens these file types using the text editor specified on the Editor/Debugger Preferences pane of the MATLAB Preferences dialog box. For text files, the default editor is the text editor specified in the MATLAB preferences.

editCommand = docblock(getEditorType) returns the current command to open the specified editor from a DocBlock block.

Input Arguments

**setEditorType** — File type whose editor command to set

'setEditorHTML' | 'setEditorDOC' | 'setEditorTXT'

File type whose editor command you want to set, specified as 'setEditorHTML', 'setEditorDOC', or 'setEditorTXT'.

**command** — Command to open file in editor

character vector | ''
Command to open the specified file type in an editor from the MATLAB command prompt, specified as a character vector. Use "" to reset to the default editor for that file type.

In the command, use the "%<FileName>" token to represent the full pathname to the document.

getEditorType — File type of the editor command to return

'getEditorHTML' | 'getEditorDOC' | 'getEditorTXT'

File type of the editor command to return, specified as 'getEditorHTML', 'getEditorDOC', or 'getEditorTXT'.

Output Arguments

editCommand — Command to open the editor
character vector

Command to open the editor, returned as a character vector.

Examples

Set DocBlock Text Editor

Specify Notepad as the DocBlock editor for text files.

docblock('setEditorTXT','system(''notepad "%<FileName>"'');')

Set and Get Current HTML Editor

You can use the docblock command to get the current editor.

Set your HTML editor for the DocBlock block to Mozilla Composer. The ampersand executes the command in the background.

docblock('setEditorHTML',...
    'system('/usr/local/bin/mozilla -edit "%<FileName>" &'');')

Get the current HTML editor.
htmlEd = docblock('getEditorHTML')
htmlEd =
    'system('/usr/local/bin/mozilla -edit "%<FileName>" &');'

**Reset Text Editor to Default**

Specify Notepad as the DocBlock editor for text files.

docblock('setEditorTXT','system(''notepad "%<FileName>''');')

Get the current text editor.

txtEd = docblock('getEditorTXT')
txtEd =
    'system('notepad "%<FileName>"');'

Reset the editor to the default editor.

docblock('setEditorTXT','')

**See Also**

DocBlock

**Topics**

“Use a Simulink DocBlock to Add a Comment” (Embedded Coder)

**Introduced in R2007a**
**edittime.getDisplayIssues**

Check whether model design warnings and errors is on

**Syntax**

val = edittime.getDisplayIssue

**Description**

Use the `val = edittime.getDisplayIssue` function to check whether model design errors and warnings is on.

**Examples**

**Enable model design errors and warnings via the command-line**

By default, model design errors and warnings are visible while you edit a model. You can turn them off by entering this command at the MATLAB command line.

`edittime.setDisplayIssues('off')`

You can turn them on by entering this command at the MATLAB command line.

`edittime.setDisplayIssues('on')`

You can check whether they are on by entering this command at the MATLAB command line.
val=edittime.getDisplayIssues

**Output Arguments**

val — Returns on or off

char

If edit-time checking is on, this function returns on. If edit-time checking is off, this function returns off.

**See Also**

edittime.setDisplayIssues

**Topics**

“Model Editing Environment”

**Introduced in R2019a**
**edittime.setDisplayIssues**

Detect model design errors and warnings

**Syntax**

edittime.setDisplayIssues(value)

**Description**

`edittime.setDisplayIssues(value)` function sets whether to display model design errors and warnings while you design your model. This function is equivalent to selecting *Display > Errors & Warnings*. This setting persists for all models.

**Examples**

**Enable model design errors and warnings through the command-line**

By default, model design errors and warnings are visible while you edit a model. You can turn them off. At the MATLAB command line, enter:

```
edittime.setDisplayIssues('off')
```

To turn on errors and warnings, at the MATLAB command line, enter:

```
edittime.setDisplayIssues('on')
```

**Input Arguments**

`value` — Enable model design warning and error detection

character vector

To enable warnings and errors while you edit your model, set `value` to `on`. To disable warnings and errors while you edit, set `value` to `off`. 

2-274
Data Types: char

See Also
edittime.getDisplayIssues

Topics
“Model Editing Environment”

Introduced in R2019a
export

(Not recommended) Export project to zip

Note simulinkproject is not recommended. Use currentProject or openProject instead. For more information, see “Compatibility Considerations”.

Syntax

export(proj, zipFileName)
export(proj, zipFileName, definitionType)

Description

export(proj, zipFileName) exports the project proj to a zip file specified by zipFileName. The zip archive preserves the project files, structure, labels, and shortcuts, and does not include any source control information. You can use the zip archive to send the project to customers, suppliers, or colleagues who do not have access to your source control repository. Recipients can create a new project from the zip archive by clicking New in the Project Tool, and then in the start page, clicking Archive.

export(proj, zipFileName, definitionType) exports the project using the specified definitionType for the project definition files, single or multiple. If you do not specify definitionType, the project's current setting is used. Use the definitionType export option if you want to change project definition file management from the type selected when the project was created. You can control project definition file management in the preferences.

Examples

Export a Project to a Zip File

Open the airframe project and create a project object.
sldemo_slproject_airframe;
proj = simulinkproject;

Export the project to a zip file.
export(proj,'airframeproj.zip')

**Input Arguments**

**proj — Project**

project object

Project, specified as a project object. Use `simulinkproject` to create a project object to manipulate a project at the command line.

**zipFileName — Zip file name or path**

character vector

Zip file name or path, specified as a character vector ending in the file extension `.zip`. If `zipFileName` is a filename, Simulink exports the file to the current folder. You can also specify a fully qualified path name.

Example: `'project.zip'`

Data Types: char

**definitionType — Definition file type**

`slproject.DefinitionFiles.SingleFile` | `slproject.DefinitionFiles.MultiFile`

Definition file type, specified as `slproject.DefinitionFiles.SingleFile`, `slproject.DefinitionFiles.MultiFile`, or `slproject.DefinitionFiles.FixedPathMultiFile`. Use the `definitionType` export option if you want to change project definition file management from the type selected when the project was created. `MultiFile` is better for avoiding merging issues on shared projects. `SingleFile` is faster but is likely to cause merge issues when two users submit changes in the same project to a source control tool. If you need to work with long file paths, use `FixedPathMultiFile`.

Example: `export(proj,'proj.zip',slproject.DefinitionFiles.SingleFile)`
Compatibility Considerations

Simulink project API is not recommended

Not recommended starting in R2019a

Starting in R2019a, instead of simulinkproject and related functions, use the currentProject or openProject functions in MATLAB. The Simulink project API will continue to be supported, but, after R2019a, new features might be available only if you use the new MATLAB project API. There are no plans to remove the Simulink project API at this time.

The new MATLAB project API is part of the R2019a functionality enabling you to use projects in MATLAB, with or without Simulink. You can now share projects with users who do not have Simulink.

See Also

Topics

“Archive Projects”

Introduced in R2013a
**findCategory**

(Not recommended) Get project category of labels

**Note**  ```simulinkproject is not recommended. Use currentProject or openProject instead. For more information, see “Compatibility Considerations”.

**Syntax**

```matlab
category = findCategory(proj, categoryName)
```  

**Description**

`category = findCategory(proj, categoryName)` returns the project category specified by `categoryName`. You need to get a category before you can use `createLabel` or `removeLabel`.

**Examples**

**Get a Category of Project Labels**

Open the airframe project and create a project object.

```matlab
sldemo_slproject_airframe;
proj = simulinkproject;
```

Use `findCategory` to get a category of labels by name.

```matlab
category = findCategory(proj, 'Classification')
category =

Category with properties:

Name: 'Classification'
```
Alternatively, you can examine categories by index. Get the first category.

proj.Categories(1)

ans =

    Category with properties:

        Name: 'Classification'
        DataType: 'none'
    LabelDefinitions: [1x8 slproject.LabelDefinition]

Find out what you can do with the category.

methods(category)

Methods for class slproject.Category:

createLabel    findLabel    removeLabel

**Input Arguments**

proj — Project
project

Project, specified as a project object. Use `simulinkproject` to create a project object to manipulate a project at the command line.

categoryName — Name of category
character vector

Name of the category to get, specified as a character vector.

**Output Arguments**

category — Category of labels
category object
Category of labels, returned as a category object that you can query or modify. If the specified category is not found, the function returns an empty array.

### Compatibility Considerations

#### Simulink project API is not recommended

*Not recommended starting in R2019a*

Starting in R2019a, instead of `simulinkproject` and related functions, use the `currentProject` or `openProject` functions in MATLAB. The Simulink project API will continue to be supported, but, after R2019a, new features might be available only if you use the new MATLAB project API. There are no plans to remove the Simulink project API at this time.

The new MATLAB project API is part of the R2019a functionality enabling you to use projects in MATLAB, with or without Simulink. You can now share projects with users who do not have Simulink.

### See Also

**Functions**

- `createLabel`
- `removeLabel`
- `simulinkproject`

**Introduced in R2013a**
**findFile**

(Not recommended) Get project file by name

**Note** simulinkproject is not recommended. Use currentProject or openProject instead. For more information, see “Compatibility Considerations”.

**Syntax**

```matlab
file = findFile(proj,fileorfolder)
```

**Description**

`file = findFile(proj,fileorfolder)` returns a specific project file by name. You need to get a file before you can query labels, or use `addLabel` or `removeLabel`.

**Examples**

**Find a File By Name**

Open the airframe project and create a project object.

```matlab
sldemo_slproject_airframe;
proj = simulinkproject;
```

Use `findFile` to get a file by name. You need to know the path if it is inside subfolders under the project root.

```matlab
myfile = findFile(proj,'models/AnalogControl.slx')
```

```matlab
myfile =
    ProjectFile with properties:
      Path: [1x86 char]
```
Alternatively, you can examine files by index. Get the first file.

```matlab
file = proj.Files(1);
```

Find out what you can do with the file.

```matlab
methods(file)
```

Methods for class `slproject.ProjectFile`:

- `addLabel`
- `removeLabel`
- `findLabel`

### Input Arguments

**proj** — Project

*project*

Project, specified as a project object. Use `simulinkproject` to create a project object to manipulate a project at the command line.

**fileorfolder** — Path of file or folder

*character vector | cell array of character vectors | string array*

Path of the file or folder to find relative to the project root folder, specified as a character vector, string, or array. Files must include any subfolders under the project root and the file extension. The file or folder must be within the root folder.

Example: `‘models/myModelName.slx’`

### Output Arguments

**file** — Project file

*file object*

Project file, returned as a file object that you can query or modify.
Compatibility Considerations

Simulink project API is not recommended
*Not recommended starting in R2019a*

Starting in R2019a, instead of `simulinkproject` and related functions, use the `currentProject` or `openProject` functions in MATLAB. The Simulink project API will continue to be supported, but, after R2019a, new features might be available only if you use the new MATLAB project API. There are no plans to remove the Simulink project API at this time.

The new MATLAB project API is part of the R2019a functionality enabling you to use projects in MATLAB, with or without Simulink. You can now share projects with users who do not have Simulink.

See Also

**Functions**
`addLabel` | `findCategory` | `findLabel` | `removeLabel` | `simulinkproject`

**Introduced in R2013a**
findLabel

(Not recommended) Get project file label

**Note** simulinkproject is not recommended. Use currentProject or openProject instead. For more information, see “Compatibility Considerations”.

**Syntax**

```matlab
label = findLabel(file, categoryName, labelName)
label = findLabel(file, labelDefinition)
label = findLabel(category, labelName)
```

**Description**

- `label = findLabel(file, categoryName, labelName)` returns the label and its attached data for the label `labelName` in the category `categoryName` that is attached to the specified file or files. Use this syntax when you know the label name and category.

- `label = findLabel(file, labelDefinition)` returns the file label and its attached data for the label name and category specified by `labelDefinition`. Use this syntax if you previously got a `labelDefinition` by accessing a `Labels` property, e.g., using a command like `myfile.Labels(1)`.

- `label = findLabel(category, labelName)` returns the label definition of the label in this category specified by `labelName`. Returns an empty array if the label is not found.

**Examples**

**Find Files with the Label Utility**

Find all project files with a particular label.

Open the airframe project and create a project object.
sldemo_slproject_airframe;
proj = simulinkproject;

Get the list of project files.

files = proj.Files;

Loop through each file. If the file has the extension .m, attach the label Utility.

for fileIndex = 1:numel(files)
    file = files(fileIndex);
    [~, ~, fileExtension] = fileparts(file.Path);
    if strcmp(fileExtension,'.m')
        addLabel(file,'Classification','Utility');
    end
end

Find all the files with the label Utility and add them to a list returned in utility_files_to_review.

utility_files_to_review = {};
for jj=1:numel(files)
    this_file = files(jj);
    label = findLabel(this_file,'Classification','Utility');

    if ( ~isempty(label))
        % This is a file labeled 'Utility'. Add to the % list of utility files.
        utility_files_to_review = [utility_files_to_review; this_file];
    end
end

**Find a Label by Name or Definition**

Open the airframe project and create a project object.

sldemo_slproject_airframe;
proj = simulinkproject;

Get a particular file by name.

myfile = findFile(proj,'models/AnalogControl.slx');
Get a label by name.

```matlab
label = findLabel(myfile,'Classification','Design');
```

Alternatively, examine the `Labels` property of the file to get an array of Label objects, one for each label attached to the file.

```matlab
labels = myfile.Labels
```

Index into the `Labels` property to get the label attached to the particular file.

```matlab
labeldefinition = myfile.Labels(1)
```

After you get the label definition from the `Labels` property, you can use it with `findLabel`.

```matlab
label = findLabel(myfile,labeldefinition);
```

### Find Labels by Name or Definition

Open the airframe project and create a project object.

```matlab
sldemo_slproject_airframe;
proj = simulinkproject;
```

Get a category.

```matlab
category = proj.Categories(1)
category =
```

```matlab
Category with properties:
```
```matlab
    Name: 'Classification'
    DataType: 'none'
    LabelDefinitions: [1x8 slproject.LabelDefinition]
```

Get a label definition.

```matlab
ld = findLabel(category,'Design')
ld =
```
```matlab
LabelDefinition with properties:
```
Name: 'Design'
CategoryName: 'Classification'

Input Arguments

file — File to search labels of
file object

File to search the labels of, specified as a file object or an array of file objects. You can get the file object by examining the project's Files property (proj.Files), or use findFile to get a file by name. The file must be in the project.

categoryName — Name of category
character vector

Name of the parent category for the label, specified as a character vector.

labelName — Name of label
character vector

Name of the label to get, specified as a character vector.

labelDefinition — Name of label
label definition object

Name of the label to get, specified as a label definition object returned by the file.Label property.

category — Category of labels
category object

Category of labels, specified as a category object. Get a category object from the proj.Categories property or by using findCategory.

Output Arguments

label — Label
label object
Label, returned as a label object.

Compatibility Considerations

Simulink project API is not recommended

*Not recommended starting in R2019a*

Starting in R2019a, instead of `simulinkproject` and related functions, use the `currentProject` or `openProject` functions in MATLAB. The Simulink project API will continue to be supported, but, after R2019a, new features might be available only if you use the new MATLAB project API. There are no plans to remove the Simulink project API at this time.

The new MATLAB project API is part of the R2019a functionality enabling you to use projects in MATLAB, with or without Simulink. You can now share projects with users who do not have Simulink.

See Also

**Functions**

`addLabel` | `createLabel` | `findFile` | `simulinkproject`

**Introduced in R2013a**
find_mdlrefs

Find Model blocks and referenced models at all levels or at top level only

Syntax

[refMdls,mdlBlks] = find_mdlrefs(system)
[refMdls,mdlBlks] = find_mdlrefs(system,Name,Value)
[refMdls,mdlBlks] = find_mdlrefs(system,allLevels)

Description

[refMdls,mdlBlks] = find_mdlrefs(system) finds all referenced models and Model blocks contained by the subsystem or model hierarchy that system is the top level of.

[refMdls,mdlBlks] = find_mdlrefs(system,Name,Value) finds referenced models and Model blocks with additional options specified by one or more Name,Value pair arguments.

[refMdls,mdlBlks] = find_mdlrefs(system,allLevels) specifies the levels of the system to search.

Tip The find_mdlrefs function provides two different ways to specify the levels of the system to search. Both techniques give the same results, but only the name and value technique allows you to control inclusion of protected models in refMdls.

Examples

Find Referenced Models in Model Hierarchy

Find referenced models and Model blocks for all models referenced by the specified model.
load_system('sldemo_mdlref_basic');
[myModels,myModelBlks] = find_mdlrefs('sldemo_mdlref_basic')

myModels = 2x1 cell array
    {'sldemo_mdlref_counter'}
    {'sldemo_mdlref_basic'}

myModelBlks = 3x1 cell array
    {'sldemo_mdlref_basic/CounterA'}
    {'sldemo_mdlref_basic/CounterB'}
    {'sldemo_mdlref_basic/CounterC'}

Input Arguments

system — System to search
character vector | handle

System to search, specified as a character vector or a handle.

• The character vector can be the path to a Model block, subsystem, or a model in a model hierarchy.
• The handle can be for a Model block, subsystem, or model in a model hierarchy.

allLevels — Levels to search
true (default) | false

Levels to search, specified as true or false.

• true — Search all Model blocks in the model hierarchy for which the system is the top model.
• false — Search only the top-level system.

Data Types: logical

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside quotes.
You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

Example: refModels = find_mdlrefs(top_model,'IncludeProtectedModels',true)

**AllLevels — Levels to search**
true (default) | false

Levels to search, specified as a true or false.

- **true** — Search all Model blocks in the model hierarchy for which the system is the top model.
- **false** — Search only the top-level system.

Data Types: logical

**IncludeProtectedModels — Include protected models in search results**
false (default) | true

Include protected models in search, specified as true or false. This setting does not affect the list of Model blocks returned.

Data Types: logical

**Variants — Include variant models in search**
'ActivePlusCodeVariants' (default) | 'ActiveVariants' | 'AllVariants'

Include variant models in search, specified as 'ActivePlusCodeVariants', 'ActiveVariants', or 'AllVariants'.

- **'ActivePlusCodeVariants'** — Include all variant models in Variant Subsystem blocks for which you select the Analyze all choices during update diagram and generate preprocessor conditionals option.
- **'ActiveVariants'** — Include active variant models for Variant Subsystem blocks.
- **'AllVariants'** — Include all variant models in Variant Subsystem blocks.

**IncludeCommented — Include commented blocks in search**
false (default) | true

Include commented blocks in search, specified as false or true.

Data Types: logical
Keep Models Loaded — Keep loaded models that function loads
false (default) | true

The `find_mdlrefs` function loads the models in the model hierarchy of the model that you specify. By default, the function closes those models, except for models that were already loaded before execution of the function. To keep all the models loaded that the function loads, set this argument to `true`.

Data Types: logical

Output Arguments

`refMdlS` — Names of referenced models
cell array of character vectors

Names of referenced models, returned as a cell array of character vectors. The last element is the system you specified in the `system` input argument or the parent model of that system.

`mdlBlks` — Names of Model blocks
cell array of character vectors

Names of Model blocks, returned as a cell array of character vectors.

See Also

Model | `find_system`

Topics

“Model Reference Basics”
“Inspect Model Hierarchies”
“Reference Protected Models from Third Parties”

Introduced before R2006a
**find_system**

Find systems, blocks, lines, ports, and annotations

**Syntax**

Objects = find_system
Objects = find_system(System)
Objects = find_system(Name,Value)
Objects = find_system(System,Name,Value)

**Description**

Objects = find_system returns loaded systems and their blocks, including subsystems.

Objects = find_system(System) returns the specified system and its blocks.

Objects = find_system(Name,Value) returns loaded systems and the objects in those systems that meet the criteria specified by one or more Name,Value pair arguments. You can use this syntax to specify search constraints and to search for specific parameter values. Specify the search constraints before the parameter and value pairs.

Objects = find_system(System,Name,Value) returns the objects in the specified system that meet the specified criteria.

**Examples**

**Find Loaded Systems and Their Blocks**

Return the names of all loaded systems and their blocks.

```matlab
load_system('vdp')
find_system
```
Returns loaded systems and libraries, including vdp.

**Find Specific System and Its Blocks**

Return vdp system and its blocks.

```matlab
load_system({'vdp','fuelsys'})
find_system('vdp')
```

ans = 13x1 cell array

```
{'vdp'    }
n{'vdp/Fcn' }
n{'vdp/More Info' }
n{'vdp/More Info/Model Info'}
n{'vdp/Mu'  }
n{'vdp/Mux' }
n{'vdp/Product' }
n{'vdp/Scope' }
n{'vdp/Sum'  }
n{'vdp/x1'  }
n{'vdp/x2'  }
n{'vdp/Out1' }
n{'vdp/Out2' }
```
**Return Names of Loaded Models**

Return the names of only the loaded models, i.e., block diagrams. This command returns library names as well, because libraries are treated as models.

```matlab
load_system('vdp');
open_bd = find_system('type','block_diagram')
```

```matlab
open_bd = 1x1 cell array
    {'vdp'}
```

**Search Children of Subsystem**

Return the names of all Goto blocks that are children of the Unlocked subsystem in the `sldemo_clutch` system.

```matlab
load_system('sldemo_clutch');
find_system('sldemo_clutch/Unlocked','SearchDepth',1,'BlockType','Goto')
```

```matlab
ans = 2x1 cell array
    {'sldemo_clutch/Unlocked/Goto' }
    {'sldemo_clutch/Unlocked/Goto1'}
```

**Search Using Multiple Criteria**

Search in the `vdp` system and return the names of all Gain blocks whose Gain value is set to 1.

```matlab
load_system('vdp');
find_system('vdp','BlockType','Gain','Gain','1')
```

```matlab
ans = 1x1 cell array
    {'vdp/Mu'}
```
**Return Lines and Annotations as Handles**

Get the handles of all lines and annotations in the vdp system. With 'FindAll', the function returns handles regardless of how you specify the system to search.

```matlab
load_system('vdp');
L = find_system('vdp','FindAll','on','type','line')
L = 17×1
  30.0042  
  29.0042  
  28.0042  
  27.0042  
  26.0042  
  25.0042  
  24.0042  
  23.0042  
  22.0042  
  21.0042  
  ...  

A = find_system('vdp','FindAll','on','type','annotation')
A = 2×1
  32.0042  
  31.0042
```

**Search for Specific Block Parameter Value**

Find any block dialog box parameters with a value of 0 in the vdp and fuelsys systems.

```matlab
load_system({'vdp','fuelsys'})
find_system({'vdp','fuelsys'},'BlockDialogParams','0')
ans =
  'vdp/x2'
  'vdp/Out1'
  'vdp/Out2'
  'fuelsys/BlockDialogParams1'
  'fuelsys/BlockDialogParams2'
```

2-297
Search Using Regular Expressions

Find all blocks in the top level of the currently loaded systems with a block dialog parameter value that starts with 3.

```matlab
load_system({'fuelsys','vdp'});
find_system('SearchDepth','1','regexp','on','BlockDialogParams','^3')
```

```matlab
ans = 3x1 cell array
    {'vdp/Scope'
    {'vdp/Scope'
    {'fuelsys/Nominal...'
```

Regular Expression Search for Partial Match

When you search using regular expressions, you can specify a part of the character vector you want to match to return all objects that contain that character vector. Find all the import and output blocks in the `sldemo_clutch` model.

```matlab
load_system('sldemo_clutch');
find_system('sldemo_clutch','regexp','on','blocktype','port')
```

```matlab
ans = 39x1 cell array
    {'sldemo_clutch/Friction...'                           }
    {'sldemo_clutch/Friction...'                           }
    {'sldemo_clutch/Friction...'                           }
    {'sldemo_clutch/Friction Mode Logic/Tin'                }
    {'sldemo_clutch/Friction Mode Logic/Tfmaxs'             }
```
Update Library Links in a Subsystem

In this example, myModel contains a single subsystem, which is a library link. After the model was last opened, a Gain block was added to the corresponding subsystem in the library.

Open the model. Use find_system with 'FollowLinks' set to 'off'. The command does not follow the library links into the subsystem and returns only the subsystem at the top level.

```matlab
open_system('myModel')
find_system(bdroot,'LookUnderMasks','on','FollowLinks', 'off')
```
Use `find_system` with 'FollowLinks' set to 'on'. `find_system` updates the library links and returns the block in the subsystem.

```matlab
find_system(bdroot,'LookUnderMasks','on','FollowLinks','on')
```

Updating Link: myModel/Subsystem/Gain
Updating Link: myModel/Subsystem/Gain

```matlab
ans =
    'myModel'
    'myModel/Subsystem'
    'myModel/Subsystem/Gain'
```

### Return Values as Handles

Provide the system to `find_system` as a handle. Search for block dialog box parameters with a value of 0. If you make multiple calls to `get_param` for the same block, then using the block handle is more efficient than specifying the full block path as a character vector.

```matlab
load_system('vdp');
sys = get_param('vdp','Handle');
find_system(sys,'BlockDialogParams','0')
```

```matlab
ans = 8×1
     3.0001
     4.0001
     8.0001
    11.0001
    12.0001
    12.0001
    13.0001
    13.0001
```
**Input Arguments**

**System — System to search**
- path name | cell array of path names | handle | vector of handles

System to search, specified as the full system path name, a cell array of system path names, a handle, or a vector of handles.

Example: 'MyModel/Subsystem1'
Example: {'vdp','fuelsys'}

**Name-Value Pair Arguments**

Specify optional comma-separated pairs of **Name**, **Value** arguments. **Name** is the argument name and **Value** is the corresponding value. **Name** must appear inside quotes. You can specify several name and value pair arguments in any order as **Name1,** **Value1,** ..., **NameN,** **ValueN**.

When you use the `find_system` function, **Name,** **Value** pair arguments can include search constraints and parameter name and value pairs. You can specify search constraints in any order, but they must precede the parameter name and value pairs.

See “Model Parameters” on page 6-2 and “Block-Specific Parameters” on page 6-132 for the lists of model and block parameters.

Example: 'SearchDepth', '0', 'LookUnderMasks', 'none', 'BlockType', 'Goto' searches in loaded systems, excluding masked subsystems, for Goto blocks.

**BlockDialogParams — Option to search block dialog box parameters for the specified value**
- character vector | string scalar

Option to search block dialog box parameters for the specified value, specified as the comma-separated pair consisting of 'BlockDialogParams' and a character vector or string scalar. This pair must follow the other search constraint pairs.

**CaseSensitive — Option to consider case when matching**
- 'on' (default) | 'off'

Option to consider case when matching, specified as the comma-separated pair consisting of 'CaseSensitive' and 'on' for case-sensitive searching or 'off'.
FindAll — Option to include lines, ports, and annotations within systems

'off' (default) | 'on'

Option to include lines, ports, and annotations in systems in the search, specified as the comma-separated pair consisting of 'FindAll' and 'on' or 'off'. When this option is set to 'on', find_system returns a vector of handles regardless of how you specify the System argument.

FirstResultOnly — Option to return only the first result

'off' (default) | 'on'

Option to return only the first result and then stop the search, specified as the comma-separated pair consisting of 'FirstResultOnly' and 'on' or 'off'.

DiveInsideSubsystemReference — Option to dive inside a referenced subsystem

'off' (default) | 'on'

Option to dive inside a referenced subsystem in a model and list child blocks specified as the comma-separated pair consisting of 'DiveInsideSubsystemReference' and 'on' or 'off'.

FollowLinks — Option to follow links into library blocks

'off' (default) | 'on'

Option to follow links into library blocks, specified as the comma-separated pair consisting of 'FollowLinks' and 'on' or 'off'. If you do not specify a system to search, find_system includes loaded libraries in the results, whether you set 'FollowLinks' to 'on' or 'off'. You can use 'FollowLinks' with 'LookUnderMasks' to update library links in subsystems. See “Update Library Links in a Subsystem” on page 2-299.

IncludeCommented — Option to include commented blocks

'off' (default) | 'on'

Option to include commented blocks in the search, specified as the comma-separated pair consisting of 'IncludeCommented' and 'on' or 'off'.

LoadFullyIfNeeded — Option to load any partially loaded models

'off' (default) | 'on'
Option to load any partially loaded models, specified as the comma-separated pair of 'LoadFullyIfNeeded' and 'on' to load models or 'off' to disable loading. Use this option, for example, to prevent load warnings.

**LookUnderMasks — Options for searching under masks**
'graphical' (default) | 'none' | 'functional' | 'all' | 'on' | 'off'

Options for searching under masks, specified as the comma-separated pair consisting of 'LookUnderMasks' and one of these options:

- 'graphical' — Search includes masked subsystems that have no workspaces and no dialogs.
- 'none' — Search skips masked subsystems.
- 'functional' — Search includes masked subsystems that do not have dialogs.
- 'all' — Search includes all masked subsystems.
- 'on' — Search includes all masked subsystems.
- 'off' — Search skips masked subsystem.

**RegExp — Option to treat search expressions as regular expressions**
'off' (default) | 'on'

Option to treat search expressions as regular expressions specified as the comma-separated pair consisting of 'RegExp' and 'on' to treat search expressions as regular expressions or 'off'. To learn more about MATLAB regular expressions, see "Regular Expressions" (MATLAB).

**'SearchDepth — Option to restrict the search depth**
positive integer character vector or string scalar

Option to restrict the search depth to the specified level, specified as the comma-separated pair consisting of 'SearchDepth' and a positive integer character vector or string scalar. For example, specify '0' to search loaded systems only, '1' for blocks and subsystems of the top-level system, '2' for the top-level system and its children, etc. The default is to search all levels.

**Variants — Options for searching variants**
'ActiveVariants' (default) | 'AllVariants' | 'ActivePlusCodeVariants'

Options for searching variants, specified as the comma-separated pair consisting of 'Variants' and one of these options:
• 'ActiveVariants' — Search in only the active variant subsystems.
• 'AllVariants' — Search in all variants.
• 'ActivePlusCodeVariants' — Search all variants if any generate preprocessor conditionals. Otherwise, search only the active variant.

This search constraint applies only to variant subsystems and model variants.

**Output Arguments**

**Objects — Matching objects**
cell array of path names | vector of handles

Matching objects found, returned as:

• A cell array of path names if you specified System as a path name or cell array of path names, or if you did not specify a system
• A vector of handles if you specified System as a handle or vector of handles

**See Also**
Simulink.allBlockDiagrams | Simulink.findBlocks | Simulink.findBlocksOfType | find_mdlrefs | getSimulinkBlockHandle | get_param | set_param

**Topics**
“Find Model Elements in Simulink Models”
“Edit and Manage Workspace Variables by Using Model Explorer”
“Regular Expressions” (MATLAB)
“Model Parameters” on page 6-2
“Block-Specific Parameters” on page 6-132

**Introduced before R2006a**
fixdt

Create Simulink.NumericType object describing fixed-point or floating-point data type

Syntax

a = fixdt(Signed, WordLength)
a = fixdt(Signed, WordLength, FractionLength)
a = fixdt(Signed, WordLength, TotalSlope, Bias)
a = fixdt(Signed, WordLength, SlopeAdjustmentFactor, FixedExponent, Bias)
a = fixdt(DataTypeNameString)
a = fixdt(..., 'DataTypeOverride', 'Off')
[DataType,IsScaledDouble] = fixdt(DataTypeNameString)
[DataType,IsScaledDouble] = fixdt(DataTypeNameString,
'DataTypeOverride', 'Off')

Description

a = fixdt(Signed, WordLength) returns a Simulink.NumericType object describing a fixed-point data type with unspecified scaling. The scaling would typically be determined by another block parameter. Signed can be 0 (false) for unsigned or 1 (true) for signed.

a = fixdt(Signed, WordLength, FractionLength) returns a Simulink.NumericType object describing a fixed-point data type with binary point scaling. FractionLength can be greater than WordLength. For more information, see “Binary Point Interpretation” (Fixed-Point Designer).

a = fixdt(Signed, WordLength, TotalSlope, Bias) or a = fixdt(Signed, WordLength, SlopeAdjustmentFactor, FixedExponent, Bias) returns a Simulink.NumericType object describing a fixed-point data type with slope and bias scaling.

a = fixdt(DataTypeNameString) returns a Simulink.NumericType object describing an integer, fixed-point, or floating-point data type specified by a data type
name. The data type name can be either the name of a built-in Simulink data type or the name of a fixed-point data type that conforms to the naming convention for fixed-point names established by the Fixed-Point Designer product. For more information, see “Fixed-Point Data Type and Scaling Notation” (Fixed-Point Designer).

\[
a = \text{fixdt}(..., 'DataTypeOverride', 'Off')
\]
returns a Simulink.NumericType object with its DataTypeOverride parameter set to Off. The default value for this property is Inherit. You can specify the DataTypeOverride parameter after any combination of other input parameters.

\[
[\text{DataType}, \text{IsScaledDouble}] = \text{fixdt}(\text{DataTypeNameString})
\]
returns a Simulink.NumericType object describing an integer, fixed-point, or floating-point data type specified by a data type name and a flag that indicates whether the specified data type name was the name of a scaled double data type.

\[
[\text{DataType}, \text{IsScaledDouble}] = \text{fixdt}(\text{DataTypeNameString}, '\text{DataTypeOverride}', 'Off')
\]
returns:

- A Simulink.NumericType object describing an integer, fixed-point, or floating-point data type specified by a data type name. The DataTypeOverride parameter of the Simulink.NumericType object is set to Off.
- A flag that indicates whether the specified data type name was the name of a scaled double data type.

### Examples

Return a Simulink.NumericType object describing a fixed-point data type with unspecified scaling:

\[
a = \text{fixdt}(1,16)
\]

\[
a = \\
\text{Simulink.NumericType}\n\begin{align*}
\text{DataTypeMode:} & \quad '\text{Fixed-point: unspecified scaling}' \\
\text{Signedness:} & \quad '\text{Signed}' \\
\text{WordLength:} & \quad 16 \\
\text{IsAlias:} & \quad \text{false} \\
\text{HeaderFile:} & \quad '' \\
\text{Description:} & \quad ''
\end{align*}
\]
Return a Simulink.NumericType object describing a fixed-point data type with binary point scaling:

```matlab
a = fixdt(1,16,2)
a =
```

Simulink.NumericType
  DataTypeMode: 'Fixed-point: binary point scaling'
  Signedness: 'Signed'
  WordLength: 16
  FractionLength: 2
  IsAlias: false
  HeaderFile: ''
  Description: ''

Return a Simulink.NumericType object describing a fixed-point data type with slope and bias scaling:

```matlab
a = fixdt(1, 16, 2^-2, 4)
a =
```

Simulink.NumericType
  DataTypeMode: 'Fixed-point: slope and bias scaling'
  Signedness: 'Signed'
  WordLength: 16
  Slope: 0.25
  Bias: 4
  IsAlias: false
  HeaderFile: ''
  Description: ''

Return a Simulink.NumericType object describing an integer, fixed-point, or floating-point data type specified by a data type name:

```matlab
[DataType,IsScaledDouble] = fixdt('ufix8')
```

```
DataType =
```

Simulink.NumericType
  DataTypeMode: 'Fixed-point: binary point scaling'
  Signedness: 'Unsigned'
  WordLength: 8
  FractionLength: 0
Return a Simulink.NumericType object with its DataTypeOverride property set to Off:

```matlab
a = fixdt(0, 8, 2, 'DataTypeOverride', 'Off')
```

```matlab
a =
```

Simulink.NumericType

- **DataTypeMode**: 'Fixed-point: binary point scaling'
- **Signedness**: 'Unsigned'
- **WordLength**: 8
- **FractionLength**: 2
- **DataTypeOverride**: Off
- **IsAlias**: false
- **HeaderFile**: ''
- **Description**: ''

**See Also**

- `float`
- `sfix`
- `sfrac`
- `sint`
- `ufix`
- `ufrac`
- `uint`

**Topics**

“Fixed-Point Data Type and Scaling Notation” (Fixed-Point Designer)

**Introduced before R2006a**
**fixpt_evenspace_cleanup**

Modify breakpoints of lookup table to have even spacing.

**Syntax**

\[ xdata\_modified = fixpt\_evenspace\_cleanup(xdata,xdt,xscale) \]

**Description**

\( xdata\_modified = fixpt\_evenspace\_cleanup(xdata,xdt,xscale) \) modifies breakpoints of a lookup table to have even spacing after quantization. By adjusting breakpoints to have even spacing after quantization, Simulink Coder generated code can exclude breakpoints from memory.

\( xdata \) is the breakpoint vector of a lookup table to make evenly spaced, such as \( 0:0.005:1 \). \( xdt \) is the data type of the breakpoints, such as \( \text{sfix}(16) \). \( xscale \) is the scaling of the breakpoints, such as \( 2^{-12} \). Using these three inputs, \( \text{fixpt\_evenspace\_cleanup} \) returns the modified breakpoints in \( xdata\_modified \).

This function works only with nontunable data and considers data to have even spacing relative to the scaling slope. For example, the breakpoint vector \([0 \ 2 \ 5]\), which has spacing value 2 and 3, appears to have uneven spacing. However, the difference between the maximum spacing 3 and the minimum spacing 2 equals 1. If the scaling slope is 1 or greater, a spacing variation of 1 represents a 1-bit change or less. In this case, the \( \text{fixpt\_evenspace\_cleanup} \) function considers a spacing variation of 1 bit or less to be even.

Modifications to breakpoints can change the numerical behavior of a lookup table. To check for changes, test the model using simulation, rapid prototyping, or other appropriate methods.

**Examples**

Modify breakpoints of a lookup table to have even spacing after quantization:
xdata = 0:0.005:1;
xdt = sfix(16);
xscale = 2^-12;
xdata_modified = fixpt_evenspace_cleanup(xdata,xdt,xscale)

See Also
fixdt | fixpt_interpl | fixpt_look1_func_approx | fixpt_look1_func_plot | sfix | ufix

Topics
“Effects of Spacing on Speed, Error, and Memory Usage” (Fixed-Point Designer)
“Create Lookup Tables for a Sine Function” (Fixed-Point Designer)

Introduced before R2006a
fixpt_interp1

Implement 1-D lookup table

Syntax

\[ y = \text{fixpt_interp1}(xdata, ydata, x, xdt, xscale, ydt, yscale, rndmeth) \]

Description

\[ y = \text{fixpt_interp1}(xdata, ydata, x, xdt, xscale, ydt, yscale, rndmeth) \]

implements a one-dimensional lookup table to find output \( y \) for input \( x \). If \( x \) falls between two \( xdata \) values (breakpoints), \( y \) is the result of interpolating between the corresponding \( ydata \) values. If \( x \) is greater than the maximum value in \( xdata \), \( y \) is the maximum \( ydata \) value. If \( x \) is less than the minimum value in \( xdata \), \( y \) is the minimum \( ydata \) value.

If the input data type \( xdt \) or the output data type \( ydt \) is floating point, \text{fixpt_interp1} performs the interpolation using floating-point calculations. Otherwise, \text{fixpt_interp1} uses integer-only calculations. These calculations handle the input scaling \( xscale \) and the output scaling \( yscale \) and obey the rounding method \( rndmeth \).

Input Arguments

\textit{xdata}

Vector of breakpoints for the lookup table, such as \( \text{linspace}(0,8,33) \).

\textit{ydata}

Vector of table data that correspond to the breakpoints for the lookup table, such as \( \text{sin}(xdata) \).

\textit{x}

Vector of input values for the lookup table to process, such as \( \text{linspace}(-1,9,201) \).
xdt
Data type of input x, such as sfix(8).

xscale
Scaling for input x, such as 2^-3.

ydt
Data type of output y, such as sfix(16).

yscale
Scaling for output y, such as 2^-14.

rndmeth
Rounding mode supported by fixed-point Simulink blocks:

'Ceiling'    Round to the nearest representable number in the direction of positive infinity.
'Floor' (default) Round to the nearest representable number in the direction of negative infinity.
'Nearest'    Round to the nearest representable number.
'Toward Zero' Round to the nearest representable number in the direction of zero.

Examples
Interpolate outputs for x using a 1-D lookup table that approximates the sine function:

```matlab
xdata = linspace(0,8,33).';
ydata = sin(xdata);
% Define input x as a vector of 201 evenly spaced points between -1 and 9 (includes values both lower and higher than the range of breakpoints in xdata)
x = linspace(-1,9,201).';
% Interpolate output values for x
```
y = fixpt_interp1(xdata,ydata,x,sfix(8),2^-3,sfix(16),...
  2^-14,'Floor')

See Also
fixpt_evenspace_cleanup|fixpt_look1_func_approx|
fixpt_look1_func_plot

Topics
“Producing Lookup Table Data” (Fixed-Point Designer)

Introduced before R2006a
fixpt_look1_func_approx

Optimize fixed-point approximation of nonlinear function by interpolating lookup table data points

Syntax

\[
[xdata,ydata,erworst] = \text{fixpt\_look1\_func\_approx}(\text{func}',... \\\ \\\ \\ \\ x_{min},x_{max},x_{dt},x_{scale},y_{dt},y_{scale},\text{rndmeth},\text{errmax},\text{nptsmax})
\]
\[
[xdata,ydata,erworst] = \text{fixpt\_look1\_func\_approx}(\text{func}',... \\\ \\\ \\ \ x_{min},x_{max},x_{dt},x_{scale},y_{dt},y_{scale},\text{rndmeth},\text{errmax},[]) \]
\[
[xdata,ydata,erworst] = \text{fixpt\_look1\_func\_approx}(\text{func}',... \\\ \\\ \\ \ x_{min},x_{max},x_{dt},x_{scale},y_{dt},y_{scale},\text{rndmeth},[],\text{nptsmax})
\]
\[
[xdata,ydata,erworst] = \text{fixpt\_look1\_func\_approx}(\text{func}',... \\\ \\\ \\ \ x_{min},x_{max},x_{dt},x_{scale},y_{dt}y_{dt},y_{scale},\text{rndmeth},\text{errmax},\text{nptsmax},\text{spacing})
\]

Description

\[
[xdata,ydata,erworst] = \text{fixpt\_look1\_func\_approx}(\text{func}',... \\\ \\\ \\ \ x_{min},x_{max},x_{dt},x_{scale},y_{dt},y_{scale},\text{rndmeth},\text{errmax},\text{nptsmax}) \]
\text{returns the optimal breakpoints of a lookup table, an ideal function applied to the breakpoints, and the worst-case approximation error. The lookup table satisfies the maximum acceptable error and maximum number of points that you specify.}

\[
[xdata,ydata,erworst] = \text{fixpt\_look1\_func\_approx}(\text{func}',... \\\ \\\ \\ \ x_{min},x_{max},x_{dt},x_{scale},y_{dt},y_{scale},\text{rndmeth},\text{errmax},[]) \]
\text{returns the optimal breakpoints of a lookup table, an ideal function applied to the breakpoints, and the worst-case approximation error. The lookup table satisfies the maximum acceptable error that you specify.}

\[
[xdata,ydata,erworst] = \text{fixpt\_look1\_func\_approx}(\text{func}',... \\\ \\\ \\ \ x_{min},x_{max},x_{dt},x_{scale},y_{dt},y_{scale},\text{rndmeth},[],\text{nptsmax}) \]
\text{returns the optimal breakpoints of a lookup table, an ideal function applied to the breakpoints, and the worst-case approximation error. The lookup table satisfies the maximum number of points that you specify.}

\[
[xdata,ydata,erworst] = \text{fixpt\_look1\_func\_approx}(\text{func}',... \\\ \\\ \\ \ x_{min},x_{max},x_{dt},x_{scale},y_{dt}y_{dt},y_{scale},\text{rndmeth},\text{errmax},\text{nptsmax},\text{spacing})
\]
fixpt_look1_func_approx

\( \text{xmin, xmax, xdt, xscale, ydtydt, yscale, rndmeth, errmax, nptsmax, spacing} \)
returns the optimal breakpoints of a lookup table, an ideal function applied to the
breakpoints, and the worst-case approximation error. The lookup table satisfies the
maximum acceptable error, maximum number of points, and breakpoint spacing that you
specify.

In each case, fixpt_look1_func_approx interpolates between lookup table data points
to optimize the fixed-point approximation. The inputs \( \text{xmin} \) and \( \text{xmax} \) specify the range
over which to approximate the breakpoints. The inputs \( \text{xdt, xscale, ydt, yscale} \), and
rndmeth follow conventions used by fixed-point Simulink blocks.

The inputs \( \text{errmax, nptsmax} \), and \( \text{spacing} \) are optional. Of these inputs, you must
specify at least \( \text{errmax} \) or \( \text{nptsmax} \). If you omit one of those two inputs, you must use
brackets, [], in place of the omitted input. fixpt_look1_func_approx ignores that
requirement for the lookup table.

If you do not specify spacing, and more than one spacing satisfies \( \text{errmax} \) and \( \text{nptsmax} \),
fixpt_look1_func_approx chooses in this order: power-of-2 spacing, even spacing,
uneven spacing. This behavior applies when you specify both \( \text{errmax} \) and \( \text{nptsmax} \), but
not when you specify just one of the two.

**Input Arguments**

**func**

Function of x for which to approximate breakpoints. Enclose this expression in single
quotes, for example, 'sin(2*pi*x)'.

**xmin**

Minimum value of x.

**xmax**

Maximum value of x.

**xdt**

Data type of x.
**xscale**
Scaling for the x values.

**ydt**
Data type of y.

**yscale**
Scaling for the y values.

**rndmeth**
Rounding mode supported by fixed-point Simulink blocks:

- 'Ceiling' Round to the nearest representable number in the direction of positive infinity.
- 'Floor' (default) Round to the nearest representable number in the direction of negative infinity.
- 'Nearest' Round to the nearest representable number.
- 'Toward Zero' Round to the nearest representable number in the direction of zero.

**errmax**
Maximum acceptable error between the ideal function and the approximation given by the lookup table.

**nptsmax**
Maximum number of points for the lookup table.

**spacing**
Spacing of breakpoints for the lookup table:

- 'even' Even spacing
- 'pow2' Even, power-of-2 spacing
- 'unrestricted' (default) Uneven spacing
If you specify...

<table>
<thead>
<tr>
<th>The breakpoints of the lookup table...</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{errmax} and \textit{nptsmax}</td>
</tr>
<tr>
<td>Meet both criteria, if possible.</td>
</tr>
<tr>
<td>The \textit{errmax} requirement has higher priority than \textit{nptsmax}. If the breakpoints cannot meet both criteria with the specified spacing, \textit{nptsmax} does not apply.</td>
</tr>
<tr>
<td>\textit{errmax} only</td>
</tr>
<tr>
<td>Meet the error criteria, and fixpt_look1_func_approx returns the fewest number of points.</td>
</tr>
<tr>
<td>\textit{nptsmax} only</td>
</tr>
<tr>
<td>Meet the points criteria, and fixpt_look1_func_approx returns the smallest worst-case error.</td>
</tr>
</tbody>
</table>

### Output Arguments

**xdata**

Vector of breakpoints for the lookup table.

**ydata**

Vector of values from applying the ideal function to the breakpoints.

**errworst**

Worst-case error, which is the maximum absolute error between the ideal function and the approximation given by the lookup table.

### Examples

Approximate a fixed-point sine function using a lookup table:

```matlab
func = 'sin(2*pi*x)';
% Define the range over which to optimize breakpoints
xmin = 0;
xmax = 0.25;
% Define the data type and scaling for the inputs
```
xdt = ufix(16);
xscale = 2^-16;
% Define the data type and scaling for the outputs
ydt = sfix(16);
yscale = 2^-14;
% Specify the rounding method
rndmeth = 'Floor';
% Define the maximum acceptable error
errmax = 2^-10;
% Choose even, power-of-2 spacing for breakpoints
spacing = 'pow2';
% Create the lookup table
[xdata,ydata,errworst] = fixpt_look1_func_approx(func,...
    xmin,xmax,xdt,xscale,ydt,yscale,rndmeth,errmax,[],spacing);

See Also
fixpt_evenspace_cleanup | fixpt_interp1 | fixpt_look1_func_plot

Topics
“Producing Lookup Table Data” (Fixed-Point Designer)
“Use Lookup Table Approximation Functions” (Fixed-Point Designer)

Introduced before R2006a
fixpt_look1_func_plot

Plot fixed-point approximation function for lookup table

Syntax

fixpt_look1_func_plot(xdata,ydata,'func',...  
xmin,xmax,xdt,xscale,ydt,yscale,rndmeth)  
errworst = fixpt_look1_func_plot(xdata,ydata,'func',...  
xmin,xmax,xdt,xscale,ydt,yscale,rndmeth)

Description

fixpt_look1_func_plot(xdata,ydata,'func',...  
xmin,xmax,xdt,xscale,ydt,yscale,rndmeth) plots a lookup table approximation  
function and the error from the ideal function.

errworst = fixpt_look1_func_plot(xdata,ydata,'func',...  
xmin,xmax,xdt,xscale,ydt,yscale,rndmeth) plots a lookup table approximation  
function and the error from the ideal function. The output errworst is the maximum  
absolute error.

You can use fixpt_look1_func_approx to generate xdata and ydata, the  
breakpoints and table data for the lookup table, respectively.  
fixpt_look1_func_approx applies the ideal function to the breakpoints in xdata to  
produce ydata. While this method is the easiest way to generate ydata, you can choose  
other values for ydata as input for fixpt_look1_func_plot. Choosing different values  
for ydata can, in some cases, produce a lookup table with a smaller maximum absolute  
error.

Input Arguments

xdata

Vector of breakpoints for the lookup table.
**ydata**
Vector of values from applying the ideal function to the breakpoints.

**func**
Function of \( x \) for which to approximate breakpoints. Enclose this expression in single quotes, for example, \( '\sin(2*\pi*x)' \).

**xmin**
Minimum value of \( x \).

**xmax**
Maximum value of \( x \).

**xdt**
Data type of \( x \).

**xscale**
Scaling for the \( x \) values.

**ydt**
Data type of \( y \).

**yscale**
Scaling for the \( y \) values.

**rndmeth**
Rounding mode supported by fixed-point Simulink blocks:

- **'Ceiling'**
  Round to the nearest representable number in the direction of positive infinity.

- **'Floor'** (default)
  Round to the nearest representable number in the direction of negative infinity.

- **'Nearest'**
  Round to the nearest representable number.
'Towards Zero'
Round to the nearest representable number in the direction of zero.

Examples
Plot a fixed-point approximation of the sine function using data points generated by fixpt_look1_func_approx:

```matlab
func = 'sin(2*pi*x)';
x = linspace(0, 0.25, 256);
x = [x -eps(x); x +eps(x)];

% Define the range over which to optimize breakpoints
xmin = 0;
xmax = 0.25;

% Define the data type and scaling for the inputs
xdt = ufix(16);
xscale = 2^-16;

% Define the data type and scaling for the outputs
ydt = sfix(16);
yscale = 2^-14;

% Specify the rounding method
rndmeth = 'Floor';

% Define the maximum acceptable error
errmax = 2^-10;

% Choose even, power-of-2 spacing for breakpoints
spacing = 'pow2';

% Generate data points for the lookup table
[xdata,ydata,errworst]=fixpt_look1_func_approx(func,...
    xmin,xmax,xdt,xscale,ydt,yscale,rndmeth,errmax,[],spacing);
% Plot the sine function (ideal and fixed-point) & errors
fixpt_look1_func_plot(xdata,ydata,func,xmin,xmax,...
    xdt,xscale,ydt,yscale,rndmeth);
```

fixpt_look1_func_plot plots the fixed-point sine function, using generated data points, and plots the error between the ideal function and the fixed-point function. The maximum absolute error and the number of points required appear on the plot. The error drops to zero at a breakpoint, but increases between breakpoints due to curvature differences between the ideal function and the line drawn between breakpoints.
The lookup table requires 33 points to achieve a maximum absolute error of $2^{-11.3922}$.

**See Also**

fixpt_evenspace_cleanup | fixpt_interp1 | fixpt_look1_func_approx

**Topics**

“Producing Lookup Table Data” (Fixed-Point Designer)
“Use Lookup Table Approximation Functions” (Fixed-Point Designer)
Introduced before R2006a
fixpt_set_all

Set property for each fixed-point block in subsystem

Syntax

fixpt_set_all(SystemName, fixptPropertyName, fixptPropertyValue)

Description

fixpt_set_all(SystemName, fixptPropertyName, fixptPropertyValue) sets the property fixptPropertyName of every applicable block in the model or subsystem SystemName to the value fixptPropertyValue

Examples

Set each fixed-point block in a model Filter_1 to round towards the floor and saturate upon overflow:

% Round towards the floor
fixpt_set_all('Filter_1', 'RndMeth', 'Floor')

% Saturate upon overflow
fixpt_set_all('Filter_1', 'DoSatur', 'on')

Introduced before R2006a
**fixptbestexp**

Exponent that gives best precision for fixed-point representation of value

**Syntax**

```plaintext
out = fixptbestexp(RealWorldValue, TotalBits, IsSigned)
out = fixptbestexp(RealWorldValue, FixPtDataType)
```

**Description**

`out = fixptbestexp(RealWorldValue, TotalBits, IsSigned)` returns the exponent that gives the best precision for the fixed-point representation of `RealWorldValue`. `TotalBits` specifies the number of bits for the fixed-point number. `IsSigned` specifies whether the fixed-point number is signed: 1 indicates the number is signed and 0 indicates the number is not signed.

`out = fixptbestexp(RealWorldValue, FixPtDataType)` returns the exponent that gives the best precision based on the data type `FixPtDataType`.

**Examples**

Get the exponent that gives the best precision for the real-world value 4/3 using a signed, 16-bit number:

```plaintext
out = fixptbestexp(4/3,16,1)
out =
-14
```

Alternatively, specify the fixed-point data type:

```plaintext
out = fixptbestexp(4/3,sfix(16))
out =
-14
```
This shows that the maximum precision representation of 4/3 is obtained by placing 14 bits to the right of the binary point:

01.01010101010101

You can specify the precision of this representation in fixed-point blocks by setting the scaling to $2^{-14}$ or $2^{\text{fixptbestexp}(4/3,16,1)}$.

**See Also**

fixptbestprec

**Introduced before R2006a**
**fixptbestprec**

Determine maximum precision available for fixed-point representation of value

**Syntax**

```
out = fixptbestprec(RealWorldValue,TotalBits,IsSigned)
out = fixptbestprec(RealWorldValue,FixPtDataType)
```

**Description**

`out = fixptbestprec(RealWorldValue,TotalBits,IsSigned)` determines the maximum precision for the fixed-point representation of the real-world value specified by `RealWorldValue`. You specify the number of bits for the fixed-point number with `TotalBits`, and you specify whether the fixed-point number is signed with `IsSigned`. If `IsSigned` is 1, the number is signed. If `IsSigned` is 0, the number is not signed. The maximum precision is returned to `out`.

`out = fixptbestprec(RealWorldValue,FixPtDataType)` determines the maximum precision based on the data type specified by `FixPtDataType`.

**Examples**

**Example 1**

The following command returns the maximum precision available for the real-world value 4/3 using a signed, 8-bit number:

```
out = fixptbestprec(4/3,8,1)
```

```
out = 0.015625
```

Alternatively, you can specify the fixed-point data type:

```
out = fixptbestprec(4/3,sfix(8))
```
out =  
0.015625

This value means that the maximum precision available for 4/3 is obtained by placing six bits to the right of the binary point since \(2^{-6}\) equals 0.015625:

01.010101

**Example 2**

You can use the maximum precision as the scaling in fixed-point blocks. This enables you to use `fixptbestprec` to perform a type of autoscaling if you would like to designate a known range of your simulation. For example, if your known range is -13 to 22, and you are using a safety margin of 30%:

```matlab
knownMax = 22;
knownMin = -13;
localSafetyMargin = 30;
slope = max( fixptbestprec( (1+localSafetyMargin/100)* ... 
    [knownMax,knownMin], sfix(16) ) );
```

The variable `slope` can then be used in the expression that you specify for the **Output data type** parameter in a block mask. Be sure to select the **Lock output data type setting against changes by the fixed-point tools** check box in the same block to prevent the Fixed-Point Tool from overriding the scaling. If you know the range, you can use this technique in place of relying on a model simulation to provide the range to the autoscaling tool, as described in `autofixexp` in the Fixed-Point Designer documentation.

**See Also**

`fixptbestexp`

**Introduced before R2006a**
**float**

Create Simulink.NumericType object describing floating-point data type

**Syntax**

```matlab
a = float('single')
a = float('double')
```

**Description**

`a = float('single')` returns a Simulink.NumericType object that describes the data type of an IEEE single (32 total bits, 8 exponent bits).

`a = float('double')` returns a Simulink.NumericType object that describes the data type of an IEEE double (64 total bits, 11 exponent bits).

**Note** float is a legacy function. In new code, use fixdt instead. In existing code, replace `float('single')` with `fixdt('single')` and `float('double')` with `fixdt('double')`.

**Examples**

Define an IEEE single data type.

```matlab
>> a = float('single')
a =

NumericType with properties:

    DataTypeMode: 'Single'
    IsAlias: 0
    DataScope: 'Auto'
    HeaderFile: ''
    Description: ''
```
See Also
Simulink.NumericType | fixdt | sfix | sfrac | sint | ufix | ufrac | uint

Introduced before R2006a
frameedit

Edit print frames for Simulink and Stateflow block diagrams

Syntax

frameedit
frameedit filename

Description

frameedit starts the PrintFrame Editor, a graphical user interface you use to create borders for Simulink and Stateflow block diagrams. With no argument, frameedit opens the PrintFrame Editor window with a new file.

frameedit filename opens the PrintFrame Editor window with the specified filename, where filename is a figure file (.fig) previously created and saved using frameedit.

Tips

This illustrates the main features of the PrintFrame Editor.
Closing the PrintFrame Editor

To close the PrintFrame Editor window, click the close box in the upper right corner, or select Close from the File menu.

Printing Simulink Block Diagrams with Print Frames

Select Print from the Simulink File menu. Check the Frame box and supply the filename for the print frame you want to use. Click OK in the Print dialog box.
Getting Help for the PrintFrame Editor

For further instructions on using the PrintFrame Editor, select PrintFrame Editor Help from the Help menu in the PrintFrame Editor.

Introduced in R2008b
**fxptdlg**

Start Fixed-Point Tool

**Syntax**

`fxptdlg('modelname')`

**Description**

`fxptdlg('modelname')` starts the Fixed-Point Tool for the Simulink model specified by `modelname`. You can also access this tool by the following methods:

- From the **Apps** tab, under **Code Generation** click **Fixed-Point Tool**.
- From a subsystem context (right-click) menu, select **Fixed-Point Tool**.

In conjunction with Fixed-Point Designer software, the Fixed-Point Tool provides convenient access to:

- Model and subsystem parameters that control the signal logging, fixed-point instrumentation mode, and data type override. (see “Model Parameters” on page 6-2)
- Plotting capabilities that enable you to plot data that resides in the MATLAB workspace, namely, simulation results associated with Scope, To Workspace, and root-level Outport blocks, in addition to logged signal data (see “Signal Logging” in the Simulink User's Guide)
- An interactive automatic data typing feature that proposes fixed-point data types for appropriately configured objects in your model, and then allows you to selectively accept and apply the data type proposals

You can launch the Fixed-Point Tool for any system or subsystem, and the tool controls the object selected in its **System under design** pane. If Fixed-Point Designer software is installed, the Fixed-Point Tool displays the name, data type, design minimum and maximum values, minimum and maximum simulation values, and scaling of each model object that logs fixed-point data. Additionally, if a signal saturates or overflows, the tool displays the number of times saturation or overflow occurred.
Note If your model uses accelerator or rapid accelerator simulation modes, the Fixed-Point Tool changes the simulation mode to normal during range collection, and then sets it back to its original simulation mode once the range collection simulation is complete.

Overriding Fixed-Point Specifications

Most of the functionality in the Fixed-Point Tool is for use with the Fixed-Point Designer software. However, even if you do not have Fixed-Point Designer software, you can configure data type override settings to simulate a model that specifies fixed-point data types. In this mode, the Simulink software temporarily overrides fixed-point data types with floating-point data types when simulating the model.

Note If you use fi objects or embedded numeric data types in your model or workspace, you might introduce fixed-point data types into your model. You can set fipref to prevent the checkout of a Fixed-Point Designer license.

To simulate a model without using Fixed-Point Designer:

1 Enter the following at the command line.

```
set_param(gcs, 'DataTypeOverride', 'Double',...
    'DataTypeOverrideAppliesTo', 'AllNumericTypes',...
    'MinMaxOverflowLogging', 'ForceOff')
```

2 If you use fi objects or embedded numeric data types in your model, set the fipref DataTypeOverride property to TrueDoubles or TrueSingles (to be consistent with the model-wide data type override setting) and the DataTypeOverrideAppliesTo property to All numeric types.

For example, at the MATLAB command line, enter:

```
p = fipref('DataTypeOverride', 'TrueDoubles', ... 
    'DataTypeOverrideAppliesTo', 'AllNumericTypes');
```

See Also

“Propose Fraction Lengths Using Simulation Range Data” (Fixed-Point Designer) | Fixed-Point Tool
Introduced before R2006a
**gcb**

Get path name of current block

**Syntax**

```
bl = gcb
bl = gcb(sys)
```

**Description**

`bl = gcb` returns the full block path name of the current block in the current system. The current block is:

- The most recently clicked block
- The S-Function block currently executing its corresponding MATLAB function
- The block whose callback routine is being executed
- The block whose mask is being evaluated as part of the `MaskInitialization` parameter evaluation
- The last block loaded after opening a model

`bl = gcb(sys)` returns the full block path name of the current block in the specified system. Load the system first.

**Examples**

**Get Path Name of Most Recently Selected Block**

Open a model.

```
vdp
```

In the model, select a block. For example, select the Gain block. Then, enter `gcb` at the command prompt.
gcb
ans =

vdp/Mu

Get Parameters of Current Block

Open a model.

vdp

Select the Gain block.

Use the value of gcb with get_param to get the value of the Gain parameter.

```matlab
x = get_param(gcb,'Gain')
x =
1
```

Get Current Block in Specified System

Load the model.

```matlab
load_system('sldemo_fuelsys');
```

Get the current block in the fuel_rate_control subsystem.

```matlab
bl = gcb('sldemo_fuelsys/fuel_rate_control')
bl =
sldemo_fuelsys/fuel_rate_control/validate_sample_time
```

Input Arguments

- `sys` — System that contains the block
  character vector
System that contains the block, specified as a character vector.
Example: 'vdp' 'sldemo_fuelsys/fuel_rate_control'

See Also
gcbh | gcbp | gcs | get_param

Topics
“Mask Callback Code”

Introduced before R2006a
gcbh

Get handle of current block

Syntax

gcbh

Description

gcbh returns the handle of the current block in the current system.

You can use this command to identify or address blocks that have no parent system. The command should be most useful to blockset authors.

Examples

This command returns the handle of the most recently selected block.

gcbh

ans =

281.0001

See Also

gcb | gcbp | gcs | getSimulinkBlockHandle

Introduced before R2006a
**gcbp**
Get `Simulink.BlockPath` object for current block

**Syntax**

```matlab
obj = gcbp
```

**Description**

`obj = gcbp` returns the `Simulink.BlockPath` object of the current block in the current system. The current block is:

- The most recently clicked block
- The last block loaded after opening a model

The returned `Simulink.BlockPath` object is context sensitive to model hierarchy.

**Examples**

Create `Simulink.BlockPath Object for Selected Block`

Open `sldemo_mdlref_basic`.

```matlab
sldemo_mdlref_basic
```

Click a block in the model hierarchy. For example, click the block labeled `ScopeA` in the `CounterA` instance of `sldemo_mdlref_counter`. Then, enter `obj = gcbp` at the command prompt.

```matlab
obj = gcbp
```

```matlab
obj = Simulink.BlockPath
```
Package: Simulink

Block Path:
  sldemo_mdlref_basic/CounterA
  sldemo_mdlref_counter/ScopeA

Use the getBlock method to access block path strings from this object.

Methods

See Also

gcb | gcbh | gcs | get_param

Introduced in R2019a
**gcs**

Get path name of current system

**Syntax**

`gcs`

**Description**

`gcs` returns the path name of the current system. The current system is one of these:

- During editing, the system or subsystem most recently clicked or that contains the block most recently clicked
- During simulation of a system that contains an S-Function block, the system or subsystem containing the S-Function block currently being evaluated
- During callbacks, the system containing any block whose callback routine is being executed
- During evaluation of `MaskInitialization`, the system containing the block whose mask is being evaluated
- The system most recently loaded into memory with `load_system`; only the first use of `load_system` makes the model the current system

The current system is the current model or a subsystem of the current model. Use `bdroot` to get the current model.

If you close the model that contains the current system, another open or loaded system becomes the current one. Use `gcs` to find out the new current system.

To explicitly set the current system, you can either:

- Use `set_param` with the 'CurrentSystem' parameter on the root Simulink model, for example:

  ```matlab
  set_param(0,'CurrentSystem','mymodel')
  ```
- Open the model by using `open_system` or the model name at the MATLAB command prompt.

## Examples

### Get Current System That Contains a Block

Return the path of the system that contains the most recently selected block.

Open the model `sldemo_fuelsys`. Open the subsystem To Controller.

```matlab
sldemo_fuelsys
open_system('sldemo_fuelsys/To Controller')
```

Click the Rate Transition block. Get the current system.

```matlab
gcs
ans =
    'sldemo_fuelsys/To Controller'
```

### Get Current System After Loading a Model

Open the model `f14` and get the current system.

```matlab
f14
gcs
ans =
    'f14'
```

Load the model `vdp` using `load_system`. Then get the current system.

```matlab
load_system('vdp');
gcs
ans =
    'vdp'
```
To remove vdp from memory, close it. In this example, the current system becomes the open model, f14.

close_system('vdp');
gcs

ans =

    'f14'

See Also
bdroot | gcb | gcbp

Introduced before R2006a
get_param

Get parameter names and values

Syntax

ParamValue = get_param(Object,Parameter)

Description

ParamValue = get_param(Object,Parameter) returns the name or value of the specified parameter for the specified model or block object. Open or load the Simulink model first.

Tip If you make multiple calls to get_param for the same block, then specifying the block using a numeric handle is more efficient than using the full block path. Use getSimulinkBlockHandle to get a block handle.

For parameter names, see:

• “Model Parameters” on page 6-2
• “Block-Specific Parameters” on page 6-132
• “Common Block Properties” on page 6-112

Examples

Get a Block Parameter Value and a Model Parameter Value

Load the vdp model.

load_system('vdp');

Get the value for the Expression block parameter.
BlockParameterValue = get_param('vdp/Fcn','Expression')

BlockParameterValue =
    1 - u*u

Get the value for the SolverType model parameter.

SolverType = get_param('vdp','SolverType')

SolverType =
    Variable-step

**Get Root Parameter Names and Values**

Get a list of global parameter names by finding the difference between the Simulink root parameter names and the model parameter names.

RootParameterNames = fieldnames(get_param(0,'ObjectParameters'));
load_system('vdp')
ModelParameterNames = fieldnames(get_param('vdp','ObjectParameters'));
GlobalParameterNames = setdiff(RootParameterNames,ModelParameterNames)

GlobalParameterNames =
    'AutoSaveOptions'
    'CacheFolder'
    'CallbackTracing'
    'CharacterEncoding'
    ...
    'CurrentSystem'

Get the value of a global parameter.

GlobalParameterValue = get_param(0,'CurrentSystem')

GlobalParameterValue =
    vdp

**Get Model Parameter Names and Values**

Get a list of model parameters for the vdp model.
load_system('vdp')
ModelParameterNames = get_param('vdp','ObjectParameters')

ModelParameterNames =
    Name: [1x1 struct]
    Tag: [1x1 struct]
    Description: [1x1 struct]
    Type: [1x1 struct]
    Parent: [1x1 struct]
    Handle: [1x1 struct]
    . . .
    Version: [1x1 struct]

Get the current value of the ModelVersion model parameter for the vdp model.

ModelParameterValue = get_param('vdp','ModelVersion')
ModelParameterValue = 1.6

**Get All Blocks and a Parameter Value**

Get a list of block paths and names for the vdp model.

load_system('vdp')
BlockPaths = find_system('vdp','Type','Block')

BlockPaths =
    'vdp/Fcn'
    'vdp/More Info'
    'vdp/More Info/Model Info'
    'vdp/Mu'
    'vdp/Mux'
    'vdp/Product'
    'vdp/Scope'
    'vdp/Sum'
    'vdp/x1'
    'vdp/x2'
    'vdp/Out1'
    'vdp/Out2'

Get a list of block dialog parameters for the Fcn block.

BlockDialogParameters = get_param('vdp/Fcn','DialogParameters')

2-348
BlockDialogParameters =
    Expr: [1x1 struct]
    SampleTime: [1x1 struct]

Get the value for the Expr block parameter.

BlockParameterValue = get_param('vdp/Fcn','Expr')

BlockParameterValue =
    1 - u*u

**Get a Block Parameter Value Using a Block Handle**

If you make multiple calls to `get_param` for the same block, then using the block handle is more efficient than specifying the full block path as a character vector, e.g., `'vdp/Fcn'`.

You can use the block handle in subsequent calls to `get_param` or `set_param`. If you examine the handle, you can see that it contains a double. Do not try to use the number of a handle alone (e.g., `5.007`) because you usually need to specify many more digits than MATLAB displays. Instead, assign the handle to a variable and use that variable name to specify a block.

Use `getSimulinkBlockHandle` to load the `vdp` model if necessary (by specifying `true`), and get a handle to the FCN block.

fcnblockhandle = getSimulinkBlockHandle('vdp/Fcn',true);

Use the block handle with `get_param` and get the value for the Expr block parameter.

BlockParameterValue = get_param(fcnblockhandle,'Expression')

BlockParameterValue =
    1 - u*u

**Display Block Types for all Blocks in a Model**

Get a list of block paths and names for the vdp model.

load_system('vdp')
BlockPaths = find_system('vdp','Type','Block')
BlockPaths =
    'vdp/Fcn'
    'vdp/More Info'
    'vdp/More Info/Model Info'
    'vdp/Mu'
    'vdp/Mux'
    'vdp/Product'
    'vdp/Scope'
    'vdp/Sum'
    'vdp/x1'
    'vdp/x2'
    'vdp/Out1'
    'vdp/Out2'

Get the value for the BlockType parameter for each of the blocks in the vdp model.

BlockTypes = get_param(BlockPaths,'BlockType')

BlockTypes =
    'Fcn'
    'SubSystem'
    'SubSystem'
    'Gain'
    'Mux'
    'Product'
    'Scope'
    'Sum'
    'Integrator'
    'Integrator'
    'Outport'
    'Outport'

**Input Arguments**

**Object — Name or handle of a model or block, or root**

handle | character vector | cell array of character vectors | string array | 0

Handle or name of a model or block, or root, specified as a numeric handle or a character vector, a cell array of character vectors for multiple blocks, or 0 for root. A numeric handle must be a scalar. You can also get parameters of lines and ports, but you must use numeric handles to specify them.
Tip If you make multiple calls to get_param for the same block, then specifying a block using a numeric handle is more efficient than using the full block path. Use getSimulinkBlockHandle to get a block handle. Do not try to use the number of a handle alone (e.g., 5.007) because you usually need to specify many more digits than MATLAB displays. Assign the handle to a variable and use that variable name to specify a block.

Specify 0 to get root parameter names, including global parameters and model parameters for the current Simulink session.

- Global parameters include Editor preferences and Simulink Coder parameters.
- Model parameters include configuration parameters, Simulink Coder parameters, and Simulink Code Inspector™ parameters.

Example: 'vdp/Fcn'

Parameter — Parameter of model or block, or root
-character vector | string scalar

Parameter of model or block, or root, specified as a character vector or 0 for root. The table shows special cases.

<table>
<thead>
<tr>
<th>Specified Parameter</th>
<th>Result</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>'ObjectParameters'</td>
<td>Returns a structure array with the parameter names of the specified object (model, block, or root) as separate fields in the structure.</td>
<td></td>
</tr>
<tr>
<td>'DialogParameters'</td>
<td>Returns a structure array with the block dialog box parameter names as separate fields in the structure. If the block has a mask, the function instead returns the mask parameters.</td>
<td></td>
</tr>
</tbody>
</table>
**Specified Parameter**

Parameter name, e.g., 'BlockType'. Specify any model or block parameter, or block dialog box parameter.

**Result**

Returns the value of the specified model or block parameter.

If you specified multiple blocks as a cell array, returns a cell array with the values of the specified parameter common to all blocks. All of the specified blocks in the cell array must contain the parameter, otherwise the function returns an error.

**Example:** 'ObjectParameters'

**Data Types:** char

### Output Arguments

**ParamValue** — The name or value of the specified parameter for the specified model or block, or root

any data type, depending on the parameter

The name or value of the specified parameter for the specified model or block, or root. If you specify multiple objects, the output is a cell array of objects. The table shows special cases.

<table>
<thead>
<tr>
<th>Specified Parameter</th>
<th>ParamValue Returned</th>
</tr>
</thead>
<tbody>
<tr>
<td>'ObjectParameters'</td>
<td>A structure array with the parameter names of the specified object (model, block, or root) as separate fields in the structure.</td>
</tr>
<tr>
<td>'DialogParameters'</td>
<td>A structure array with the block dialog box parameter names as separate fields in the structure. If the block has a mask, the function instead returns the mask parameters.</td>
</tr>
<tr>
<td>Specified Parameter</td>
<td>ParamValue Returned</td>
</tr>
<tr>
<td>--------------------------------------------</td>
<td>--------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Parameter name, e.g., 'BlockType'</td>
<td>The value of the specified model or block parameter. If multiple blocks are specified as a cell array, returns a cell array with the values of the specified parameter common to all blocks.</td>
</tr>
</tbody>
</table>

If you get the root parameters by specifying `get_param(0,'ObjectParameters')`, then the output `ParamValue` is a structure array with the root parameter names as separate fields in the structure. Each parameter field is a structure containing these fields:

- **Type** — Parameter type values are: 'boolean', 'string', 'int', 'real', 'point', 'rectangle', 'matrix', 'enum', 'ports', or 'list'
- **Enum** — Cell array of enumeration character vector values that applies only to 'enum' parameter types
- **Attributes** — Cell array of character vectors defining the attributes of the parameter. Values are: 'read-write', 'read-only', 'read-only-if-compiled', 'write-only', 'dont-eval', 'always-save', 'never-save', 'nondirty', or 'simulation'

**See Also**

bdroot | find_system | gcb | gcs | getSimulinkBlockHandle | set_param

**Topics**

“Associating User Data with Blocks”
“Use MATLAB Commands to Change Workspace Data”
“Model Parameters” on page 6-2
“Block-Specific Parameters” on page 6-132
“Common Block Properties” on page 6-112

**Introduced before R2006a**
getActiveConfigSet

Get model's active configuration set or configuration reference

Syntax

myConfigObj = getActiveConfigSet(model)

Arguments

model

The name of an open model, or gcs to specify the current model

Description

getActiveConfigSet returns the configuration set or configuration reference (configuration object) that is the active configuration object of model.

Examples

The following example returns the active configuration object of the current model. The code is the same whether the object is a configuration set or configuration reference.

myConfigObj = getActiveConfigSet(gcs);

See Also

attachConfigSet | attachConfigSetCopy | closeDialog | detachConfigSet | getConfigSet | getConfigSets | openDialog | setActiveConfigSet

Topics

“Manage a Configuration Set”
“Manage a Configuration Reference”
Introduced before R2006a
getCallbackAnnotation

Get annotation executing callback

Syntax

ann = getCallbackAnnotation

Description

ann = getCallbackAnnotation gets the annotation from which a callback was invoked. Invoke a callback from an annotation click function. After you get the annotation, you can, for example, get text or parameters from the annotation to use someplace else in your model.

For information on click functions, see “Associate a Click Function with an Annotation”.

Examples

Click Annotation to Change Parameter Value

Invoke a callback by way of an annotation click function. This example shows how to change a parameter value on a block to the value shown on an annotation.

Open vdp. Add and position two annotations. Each annotation displays a different value.

open_system('vdp');
an1 = Simulink.Annotation('vdp/1');
an1.position = [100,300];
an2 = Simulink.Annotation('vdp/3');
an2.position = [150,300];

Assign a click function to each annotation. The click function uses getAnnotationCallback to get the annotation instance. Get the text from each annotation and use it to set the parameter on the Gain block (Mu).
an1.ClickFcn = 'ann = getCallbackAnnotation; v = ann.Text; set_param(''vdp/Mu'',''Gain'',v)'
an2.ClickFcn = 'ann = getCallbackAnnotation; v = ann.Text; set_param(''vdp/Mu'',''Gain'',v)'

Click each annotation. When you click, the gain value on the Mu block changes to the number shown on the annotation.

**Output Arguments**

ann — Annotation
Simulink.Annotation instance

Annotation, returned as a Simulink.Annotation instance.

**See Also**

Simulink.Annotation

**Topics**

“Associate a Click Function with an Annotation”

**Introduced before R2006a**
**getComponent**

Get a configuration set component

**Syntax**

```matlab
component = getComponent(cs, componentName)
```

**Description**

```matlab
component = getComponent(cs, componentName)
```
returns the specified component from a configuration set. With no component name specified, returns a list of the components contained in the configuration set.

**Examples**

**Get a Component for a Configuration Set**

Get the solver component for the active configuration set.

Get the active configuration set of the currently selected model.

```matlab
hCs = getActiveConfigSet(gcs);
```

Get the solver component of the active configuration set.

```matlab
hSolverConfig = getComponent(hCs, 'Solver');
```

**Get List of Components for an Active Configuration Set**

Get a list of the components contained in the active configuration set.

Get the active configuration set of the currently selected model.
hCs = getActiveConfigSet(gcs);

Get the list of components contained in the configuration set.

getComponent(hCs)

The code displays the names of the components at the MATLAB command line.

**Input Arguments**

- **cs** — Configuration set object
  - ConfigSet object
  - Configuration set, specified as a ConfigSet object

- **componentName** — Component name
  - character vector
  - Name of a component object, specified as a character vector. If a component name is not specified, the function displays a list of the components contained in the configuration set at the MATLAB command line.
  - Example: 'Solver'

**Output Arguments**

- **component** — Component
  - Simulink.ConfigComponent object
  - A component in the configuration set, returned as an instance of a Simulink.ConfigComponent object

**See Also**

**Topics**

- “About Configuration Sets”
- “Manage a Configuration Set”
Introduced before R2006a
getConfigSet

Get one of model's configuration sets or configuration references

Syntax

myConfigObj = getConfigSet(model, configObjName)

Arguments

model
  The name of an open model, or gcs to specify the current model
configObjName
  The name of a configuration set (Simulink.ConfigSet) or configuration reference (Simulink.ConfigSetRef)

Description

getConfigSet returns the configuration set or configuration reference (configuration object) that is attached to model and is named configObjName. If no such object exists, the function returns an empty object.

Examples

The following example returns the configuration object that is named DevConfig and attached to the current model. The code is the same whether DevConfig is a configuration set or configuration reference.

  myConfigObj = getConfigSet(gcs, 'DevConfig');
See Also
attachConfigSet | attachConfigSetCopy | closeDialog | detachConfigSet |
getActiveConfigSet | getConfigSets | openDialog | setActiveConfigSet

Topics
“Manage a Configuration Set”
“Manage a Configuration Reference”

Introduced before R2006a
getConfigSets
Get names of all of model's configuration sets or configuration references

Syntax
myConfigObjNames = getConfigSets(model)

Arguments
model
The name of an open model, or gcs to specify the current model

Description
getConfigSets returns a cell array of character vectors specifying the names of all configuration sets and configuration references (configuration objects) attached to model.

Examples
The following example obtains the names of the configuration objects attached to the current model.

myConfigObjNames = getConfigSets(gcs)

See Also
attachConfigSet | attachConfigSetCopy | closeDialog | detachConfigSet | getActiveConfigSet | getConfigSet | openDialog | setActiveConfigSet

Topics
“Manage a Configuration Set”
“Manage a Configuration Reference”

Introduced before R2006a
getfullname

Get pathname of block or line

Syntax

path=getfullname(handle)

Description

path=getfullname(handle) returns the full pathname of the block or line specified by handle.

Examples

getfullname(gcb) returns the pathname of the block currently selected in the model editor's window.

The following code returns the pathname of the line currently selected in the model editor's window.

```matlab
line = find_system(gcs, 'SearchDepth', 1, 'FindAll', 'on', ...
    'Type', 'line', 'Selected', 'on');
path = getfullname(line);
```

See Also

find_system | gcb

Introduced in R2007a
getInputString

Create comma-separated list of variables to map

Syntax

externalInputString = getInputString(inputmap,'base')

externalInputString = getInputString(inputmap, filename)

externalInputString = getInputString(inputmap)

Description

externalInputString = getInputString(inputmap,'base') creates an input character vector using the supplied mapping inputmap and the variables loaded in the base workspace ('base').

This function generates a comma-separated list of variables (input character vector) to be mapped. You can then use this list:

- As input to the sim command. Load the variables in the base workspace first.
- As input for the Configuration Parameters > Data Import/Export > Input parameter. Copy the contents of the input character vector into the text field.

This function is most useful if you have created a custom mapping.

externalInputString = getInputString(inputmap, filename) creates an input character vector using the supplied mapping inputmap and the variables defined in filename.

externalInputString = getInputString(inputmap) creates an input character vector using the signals from the most recently created mapping.

Examples
Create an input character vector from the base workspace

Create an input character vector from the base workspace and simulate a model.

Open the model

`slexAutotransRootInportsExample;`

Create signal variables in the base workspace

```matlab
Throttle = timeseries(ones(10,1)*10);
Brake = timeseries(zeros(10,1));
```

Create a mapping (inputMap) for the model.

```matlab
inputMap = getRootInportMap('model','slexAutotransRootInportsExample','signalName',{'Throttle','Brake'},'blockName',{'Throttle','Brake'});
```

Call `getInputString` with `inputMap` and 'base' as inputs.

```matlab
externalInputString = getInputString(inputMap,'base')
```

```matlab
externalInputString =
    Throttle, Brake
```

Simulate the model with the input character vector.

```matlab
sim('slexAutotransRootInportsExample','ExternalInput',externalInputString);
```

Create an external input character vector from variables in a MAT-file

Create an external input character vector from variables in a MAT-file named `input.mat`.

In a writable folder, create a MAT-file with input variables.

```matlab
Throttle = timeseries(ones(10,1)*10);
Brake = timeseries(zeros(10,1));
save('input.mat','Throttle','Brake');
```

Open the model.
slexAutotransRootInportsExample;

Create map object.

inputMap = getRootInportMap('model', ...  
'slexAutotransRootInportsExample', ...  
'signalName', {'Throttle', 'Brake'}, ...  
'blockName', {'Throttle', 'Brake'});

Get the resulting input character vector.

externalInputString = getInputString(inputMap, 'input.mat')

externalInputString =
Throttle, Brake

Load variables from the base workspace for the simulation.

load('input.mat');

Simulate the model.

sim('slexAutotransRootInportsExample', 'ExternalInput', ...  
   externalInputString);

Create an external input character vector from only an input map

Create an input character vector from only an input map vector and simulate the model.

Open the model.

slexAutotransRootInportsExample;

Create signal variables in the base workspace

Throttle = timeseries(ones(10,1)*10);
Brake    = timeseries(zeros(10,1));

Create a mapping vector for the model.

inputMap = getSlRootInportMap('model', 'slexAutotransRootInportsExample', ...  
   'MappingMode', 'BlockName', ...
Get the resulting input character vector.

```matlab
externalInputString = getInputString(inputMap)
```

Simulate the model with the input character vector.

```matlab
sim('slexAutotransRootInportsExample','ExternalInput',
    externalInputString);
```

Alternatively, if you want to input the list of variables through the Configuration Parameters dialog, copy the contents of `externalInputString` (Throttle, Brake) into the **Data Import/Export > Input** parameter. Apply the changes, and then simulate the model.

### Output Arguments

`externalInputString — External input`  
comma-separated character vector

`inputmap — Map object`  
character vector

Map object, as returned from the `getRootInportMap` or `getSlRootInportMap` functions.

`filename — Input variables`  
MAT-file name as character vector

Input variables, contained in a MAT-file. The file contains variables to map.  
Example: 'data.mat'  
Data Types: char

### Input Arguments

`filename`  
Input variables, contained in a MAT-file. The file contains variables to map.  
Example: 'data.mat'  
Data Types: char
External input, returned as a comma-separated character vector. The character vector contains root inport information that you can specify to the sim command or the Configuration Parameters > Data Import/Export > Input parameter.

See Also
getRootInportMap | getSlRootInportMap

Topics
“Map Root Inport Signal Data”

Introduced in R2013a
getRootInportMap

Create custom object to map signals to root-level inports

Syntax

map = getRootInportMap('Empty');
map = getRootInportMap(model,mdl,Name,Value);
map = getRootInportMap(inputmap,map,Name,Value);

Description

map = getRootInportMap('Empty'); creates an empty map object, map. Use this map object to set up an empty custom mapping object. Load the model before using this function. If you do not load the model first, the function loads the model to make the mapping and then closes the model afterwards.

map = getRootInportMap(model,mdl,Name,Value); creates a map object for model, mdl, with block names and signal names specified. Load the model before using this function. If you do not load the model first, the function loads the model to make the mapping and then closes the model afterwards. To create a comma-separated list of variables to map from this object, use the getInputString function.

map = getRootInportMap(inputmap,map,Name,Value); overrides the mapping object with the specified property. You can override only the properties model, blockName, and signalName. Load the model before using this function. If you do not load the model first, the function loads the model to make the mapping and then closes the model afterwards. To create a comma-separated list of variables to map from this object, use the getInputString function.

Use the getRootInportMap function when creating a custom mapping mode to map data to root-level inports. See the example file BlockNameIgnorePrefixMap.m, located in matlabroot/help/toolbox/simulink/examples, for an example of a custom mapping algorithm.
Input Arguments

Empty

Create an empty map object.

Default: none

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside quotes. You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

model

Name of model to associate with the root inport map.

Default: None

blockName

Block names of root-level input ports. The tool assigns data to ports according to the name of the root-inport block. If the tool finds a data element whose name matches the name of a root-inport block, it maps the data to the corresponding port.

The value for this argument can be:
Block name of root-level input ports.
Cell array containing multiple block names of root-level input ports.

Default: None

signalName

Signal names to be mapped. The tool assigns data to ports according to the name of the signal on the port. If the tool finds a data element whose name matches the name of a signal at a port, it maps the data to the corresponding port.

The value for this argument can be:
Signal name to be mapped.
Cell array containing multiple signal names of signals to be mapped.

**Default:** None

`inputmap`

Name of mapping object to override.

**Default:** None

## Output Arguments

**map**

Custom object that you can use to map data to root-level input port. To create a comma-separated list of variables to map from this object, use the `getInputString` function.

## Examples

### Empty Mapping Object

Create an empty custom mapping object.

```matlab
map = getRootInportMap('Empty')
map =
    1x0 InputMap array with properties:
          Type
    DataSourceName
          Destination
```

### Simple Mapping Object

Create a simple mapping object using a MATLAB time series object.

Create a time series object, `signalIn1`. 

signalIn1 = timeseries((1:10)')

Common Properties:
   Name: 'unnamed'
   Time: [10x1 double]
   TimeInfo: [1x1 tsdata.timemetadata]
   Data: [10x1 double]
   DataInfo: [1x1 tsdata.datametadata]

Create a mapping object for the time series object for the model, ex_minportsOnlyModel.

modelFile = fullfile(matlabroot,'help','toolbox','simulink','examples','ex_minportsOnlyModel');
load_system(modelFile);
map = getRootInportMap('model','ex_minportsOnlyModel','blockName','In1','signalname','signalIn1')

map =

   InputMap with properties:
       Type: 'Inport'
       DataSourceName: 'signalIn1'
       Destination: [1x1 Simulink.iospecification.Destination]

Mapping Object with Vectors

Create a mapping object using vectors of block names and signal names for the model ex_minportsOnlyModel.

Create a mapping object of vectors.

modelFile = fullfile(matlabroot,'help','toolbox','simulink','examples','ex_minportsOnlyModel');
load_system(modelFile);
map = getRootInportMap('model','ex_minportsOnlyModel','blockName',{'In1' 'In2'},'signalname',{'signalIn1' 'signalIn2'})

map =

   1x2 InputMap array with properties:
**Overriding Maps**

Create a mapping object that contains the signal `var2`, then override `var2` with `var1`.

Create a mapping object of vectors.

```matlab
% Load the model and define variables
modelFile = fullfile(matlabroot,'help','toolbox','simulink',
        'examples','ex_minportsOnlyModel');
load_system(modelFile);
modelValue = 'ex_minportsOnlyModel';
blockNameValue = 'In1';
signalNameValue = 'var2';
portType = 'Inport';

% Define var1 and override var2 with var1
signalNameToOverload = 'var1';
mapToOverload = getRootInportMap('model',modelValue,...
        'blockName',blockNameValue,...
        'signalName',signalNameToOverload)

mapToOverload =
    InputMap with properties:
        Type: 'Inport'
        DataSourceName: 'var1'
        Destination: [1x1 Simulink.iiospecification.Destination]
```

**Tips**

- Load the model before running this function.
- If your custom mapping mode similar to an existing Simulink mapping mode, consider using the `getSlRootInportMap` function instead.
See Also
getInputString | getSlRootInportMap

Topics
“Create Custom Mapping File Function”

Introduced in R2012b
getSimulinkBlockHandle

Get block handle from block path

Syntax

handle = getSimulinkBlockHandle(path)
handle = getSimulinkBlockHandle(path,true)

Description

handle = getSimulinkBlockHandle(path) returns the numeric handle of the block specified by path, if it exists in a loaded model or library. Returns -1 if the block is not found. Library links are resolved where necessary.

Use the numeric handle returned by getSimulinkBlockHandle to manipulate the block in subsequent calls to get_param or set_param. This approach is more efficient than making multiple calls to these functions using the full block path. Do not try to use the number of a handle alone (e.g., 5.007) because you usually need to specify many more digits than MATLAB displays. Assign the handle to a variable and use that variable name to specify a block. The handle applies only to the current MATLAB session.

Use getSimulinkBlockHandle to check whether a block path is valid. This approach is more efficient than calling get_param inside a try statement.

handle = getSimulinkBlockHandle(path,true) attempts to load the model or library containing the specified block path, and then checks if the block exists. No error is returned if the model or library is not found. Any models or libraries loaded this way remain in memory even if the function does not find a block with the specified path.

Examples

Get the Handle of a Block

Get the handle of the Pilot block.
load_system('f14')
handle = getSimulinkBlockHandle('f14/Pilot')

handle = 
    562.0004

You can use the handle in subsequent calls to `get_param` or `set_param`.

Load the Model and Get the Block Handle

Load the model `f14` if necessary (by specifying `true`), and get the handle of the `Pilot` block.

handle = getSimulinkBlockHandle('f14/Pilot',true)

handle = 
    562.0004

You can use the handle in subsequent calls to `get_param` or `set_param`.

Check If a Model Contains a Specific Block

Check whether the model `f14` is loaded and contains a block named `Pilot`. Valid handles are always greater than zero. If the function does not find the block, it returns -1.

valid_block_path = getSimulinkBlockHandle('f14/Pilot') > 0

valid_block_path = 
    0

The model contains the block but the model is not loaded, so this command returns 0 because it cannot find the block.
Using `getSimulinkBlockHandle` to check whether a block path is valid is more efficient than calling `get_param` inside a `try` statement.

### Input Arguments

**path — Block path name**

character vector | cell array of character vectors

Block path name, specified as a character vector or a cell array of character vectors.

Example: `'f14/Pilot'`

Data Types: `char`

### Output Arguments

**handle — Numeric handle of a block**

double | array of doubles

Numeric handle of a block, returned as a double or an array of doubles. Valid handles are always greater than zero. If the function does not find the block, it returns `-1`. If the `path` input is a cell array of character vectors, then the output is a numeric array of handles.

Data Types: `double`

### See Also

`get_param` | `set_param`

### Introduced in R2015a
getSlRootInportMap

Create custom object to map signals to root-level inports using Simulink mapping mode

Syntax

```matlab
inputMap = getSlRootInportMap('model', modelName, 'MappingMode', mappingMode, 'SignalName', signalName, 'SignalValue', signalValue)
[inputMap, hasASignal] = getSlRootInportMap('model', modelName, 'MappingMode', mappingMode, 'SignalName', signalName, 'SignalValue', signalValue)
inputMap = getSlRootInportMap('model', modelName, 'MappingMode', 'Custom', 'CustomFunction', customFunction, 'SignalName', signalName, 'SignalValue', signalValue)
[inputMap, hasASignal] = getSlRootInportMap('model', modelName, 'MappingMode', 'Custom', 'CustomFunction', customFunction, 'SignalName', signalName, 'SignalValue', signalValue)
```

Description

```matlab
inputMap = getSlRootInportMap('model', modelName, 'MappingMode', mappingMode, 'SignalName', signalName, 'SignalValue', signalValue) creates a root inport map using one of the Simulink mapping modes. Load the model before using this function. If you do not load the model first, the function loads the model to make the mapping and then closes the model afterwards. To create a comma-separated list of variables to map from this object, use the getInputString function.

[inputMap, hasASignal] = getSlRootInportMap('model', modelName, 'MappingMode', mappingMode, 'SignalName', signalName, 'SignalValue', signalValue) returns a vector of logical values specifying whether or not the root inport map has a signal associated with it. To create a comma-separated list of variables to map from this object, use the getInputString function.

inputMap = getSlRootInportMap('model', modelName, 'MappingMode', 'Custom', 'CustomFunction',
```
customfunction,'SignalName',signalname,'SignalValue',signalvalue)
creates a root inport map using a custom mapping mode specified in customfunction.
Load the model before using this function. If you do not load the model first, the function
loads the model to make the mapping and then closes the model afterwards. To create a
comma-separated list of variables to map from this object, use the getInPutString
function.

[inputMap,hasASignal] = getSlRootInportMap('model',
modelName,'MappingMode','Custom','CustomFunction',
customfunction,'SignalName',signalname,'SignalValue',signalvalue)
returns a vector of logical values specifying whether or not the root inport map has a
signal associated with it. To create a comma-separated list of variables to map from this
object, use the getInPutString function.

To map signals to root-level inports using custom mapping modes, you can use
getSlRootInport with the Root Inport Mapper dialog box custom mapping capability.

Examples

Create inport map using Simulink mapping mode

Create a vector of inport maps using a built-in mapping mode.

Throttle = timeseries(ones(10,1)*10);
Brake    = timeseries(zeros(10,1));
inputMap = getSlRootInportMap('model','slexAutotransRootInportsExample',
    'MappingMode','BlockName',
    'SignalName',{'Throttle' 'Brake'},
    'SignalValue',{Throttle Brake});

Create inport map using custom function

Create a vector of inport maps using a custom function

port1    = timeseries(ones(10,1)*10);
port2    = timeseries(zeros(10,1));
inputMap = getSlRootInportMap('model','slexAutotransRootInportsExample',
    'MappingMode','Custom',
    'CustomFunction','slexCustomMappingMyCustomMap',
    'SignalName',signalname,'SignalValue',signalvalue);
Input Arguments

`modelname` — Model name
character vector

Specify the model to associate with the root inport map.

Data Types: char

`mappingmode` — Simulink mapping mode
character vector

Specify the mapping mode to use with model name and data source. Possible values are:

- `'Index'` Assign sequential index numbers, starting at 1, to the data in the MAT-file, and map this data to the corresponding inport.
- `'BlockName'` Assign data to ports according to the name of the root-inport block. If the block name of a data element matches the name of a root-inport block, map the data to the corresponding port.
- `'SignalName'` Assign data to ports according to the name of the signal on the port. If the signal name of a data element matches the name of a signal at a port, map the data to the corresponding port.
- `'BlockPath'` Assign data to ports according to the block path of the root-inport block. If the block path of a data element matches the block path of a root-inport block, map the data to the corresponding port.
- `'Custom'` Apply mappings according to the definitions in a custom file.

Data Types: char

`customfunction` — Custom function file name
character vector

Specify name of file that implements a custom method to map signals to root-level ports. This function must be on the MATLAB path.

Data Types: char
**signalname — signal name**  
scalar | cell array of character vectors

Specify the signal name(s) of the signal to associate with the root inport map.  
Data Types: char | cell

**signalvalue — signal value**  
scalar | cell arrays

Specify the values of the signals to map to the root inport map. For the list of supported data types for the values, see “Choose a Base Workspace and MAT-File Format”.

### Output Arguments

**inputMap — input map**  
scalar | vector

Mapping object that defines the mapping of input signals to root-level ports. To create a comma-separated list of variables to map from this object, use the `getInputString` function.

**hasASignal — signal presence indicator**  
scalar | vector

A vector of logical values with the same length as `inputMap`. If the value is true the `inputMap` has a signal associated with it. If the value is false the `inputMap` does not have a signal associated with it and will use a ground value as an input  
Data Types: logical

### Tips

- Load the model before running this function.
- If your custom mapping mode is not similar to an existing Simulink mapping mode, consider using the `getRootInportMap` function instead.
See Also
getInputString | getRootInportMap

Topics
“Map Root Inport Signal Data”

Introduced in R2013b
hdllib

Display blocks that are compatible with HDL code generation

Syntax

hdllib
hdllib('off')
hdllib('html')
hdllib('librarymodel')

Description

hdllib displays the blocks that are supported for HDL code generation, and for which you have a license, in the Library Browser. To build models that are compatible with the HDL Coder software, use blocks from this Library Browser view.

If you close and reopen the Library Browser in the same MATLAB session, the Library Browser continues to show only the blocks supported for HDL code generation. To show all blocks, regardless of HDL code generation compatibility, at the command prompt, enter hdllib('off').

hdllib('off') displays all the blocks for which you have a license in the Library Browser, regardless of HDL code generation compatibility.

hdllib('html') creates a library of blocks that are compatible with HDL code generation. It generates two additional HTML reports: a categorized list of blocks (hdlblklist.html) and a table of blocks and their HDL code generation parameters (hdlsupported.html).

To run hdllib('html'), you must have an HDL Coder license.

hdllib('librarymodel') displays blocks that are compatible with HDL code generation in the Library Browser. To build models that are compatible with the HDL Coder software, use blocks from this library.
The default library name is `hdlsupported`. After you generate the library, you can save it to a folder of your choice.

To keep the library current, you must regenerate it each time that you install a new software release.

To run `hdllib('librarymodel')`, you must have an HDL Coder license.

### Examples

#### Display Supported Blocks in the Library Browser

To display blocks that are compatible with HDL code generation in the Library Browser:

```matlab
hdllib
```

```matlab
### Generating view of HDL Coder compatible blocks in Library Browser.
### To restore the Library Browser to the default Simulink view, enter "hdllib off".
```
Display All Blocks in the Library Browser

To display all blocks in the Library Browser, regardless of HDL code generation compatibility:

```matlab
hdllib('off')
```

### Restoring Library Browser to default view; removing the HDL Coder compatibility filter.
Create a Supported Blocks Library and HTML Reports

To create a library and HTML reports showing the blocks that are compatible with HDL code generation:

```
hdllib('html')
```
Create a Supported Blocks Library

To create a library that contains blocks that are compatible with HDL code generation:

```matlab
hdllib('librarymodel')
```

The `hdlsupported` block library opens.
See Also

Topics
“Show Blocks Supported for HDL Code Generation” (HDL Coder)
“View HDL-Specific Block Documentation” (HDL Coder)
“Create Simulink Model for HDL Code Generation” (HDL Coder)

Introduced in R2006b
hilite_system

Highlight block, signal line, port, or annotation

Syntax

hilite_system(obj)
hilite_system(obj,style)

Description

hilite_system(obj) highlights a block, line, port, or annotation in an open model using the default highlight style. Use hilite_system with a port to highlight the signal line attached to the port. Each use of hilite_system adds to the highlighting. Highlighting is not saved with the model.

hilite_system(obj,style) uses the specified highlighting style.

Examples

Highlight Block Using Default Highlight Style

Open the model slexAircraftExample.

slexAircraftExample

Highlight the Controller block. When you use the default highlight style, the block appears highlighted with a red outline and yellow fill.

hilite_system('slexAircraftExample/Controller')

Highlight a Block Using a Highlight Style

Open the model vdp.
Highlight the Mu block using the style 'fade'.

hilite_system('vdp/Mu','fade')

**Use Block Highlighting to Trace Generated Code**

If you have a Simulink Coder license, you can trace generated code to the corresponding source block in a model.

Open the model `f14`.

`f14`

In the model configuration parameters, in the *Solver* pane, set *Type* to *Fixed-step*.

Generate code for the model. In the *Apps* tab, under *Code Generation*, click *Embedded Coder*. The *C Code* tab appears. Click *Build*.

In an editor or in the code generation report, open a generated source or header file. As you review lines of code, note traceability tags that correspond to code of interest.

Highlight a block using a traceability tag.

`hilite_system('<Root>/Stick Input')`
Highlight a block in a subsystem.

hilite_system(''<S3>/W-gust model'')
Customize a Highlighting Style

You can customize a highlighting style by setting the 'HiliteAncestorsData' parameter on the root-level model using `set_param` in this form:

```matlab
set_param(0,'HiliteAncestorsData',hiliteData)
```

Specify `hiliteData` as a structure array that has these fields:

- 'HiliteType' — Highlighting style to customize, such as 'user1', 'debug', or 'error'.
- 'ForegroundColor' — Color for block fill.
- 'BackgroundColor' — Color for block outline.

The supported values for 'ForegroundColor' and 'BackgroundColor' are:

- 'black'
- 'white'
- 'gray'
- 'red'
- 'orange'
- 'yellow'
- 'green'
- 'darkGreen'
- 'blue'
- 'lightBlue'
- 'cyan'
- 'magenta'

Define a highlight style for 'user1', and customize the style for 'debug'.

```matlab
set_param(0,'HiliteAncestorsData',...
    struct('HiliteType','user1',...
        'ForegroundColor','darkGreen',...
        'BackgroundColor','lightBlue'));
set_param(0,'HiliteAncestorsData',...
    struct('HiliteType','debug',...
        'ForegroundColor','red',...
        'BackgroundColor','black'));
```
Use the defined style to highlight a block.

hilite_system('f14/Controller/Alpha-sensor Low-pass Filter','user1')

**Input Arguments**

**obj — Block, port, line, or annotation to highlight**

block path name | Simulink.BlockPath object | numeric handle | Simulink identifier | traceability tag

Block, port, line, or annotation to highlight, specified as:

- The full block path name
- A Simulink.BlockPath object
- A numeric handle for lines, ports, or annotations
- Simulink identifier
- A traceability tag from the comments of Simulink Coder generated code.

Using a traceability tag requires a Simulink Coder license.

The format for a traceability tag is `<system>/block`, where `system` is either:

- Root
- A unique system number assigned by Simulink during code generation

Example: 'vdp/Mu','sldemo_fuelsys/fuel_rate_control/airflow_calc','vdp:3','<Root>/Mu'

**style — Highlighting style**

'default' (default) | character vector

Highlighting style, specified as one of these values. You can customize the appearance of any of the styles. See “Customize a Highlighting Style” on page 2-393.

- 'default' — Default color scheme: red outline, yellow fill.
- 'none' — Clears the highlight.

To clear all highlighting, in the Simulink Editor, in the **Signal** tab, click **Remove Trace**.
• 'debug' — Uses default color scheme.
• 'different' — Applies red outline, white fill.
• 'error' — Uses default color scheme.
• 'fade' — Applies gray outline, white fill.
• 'find' — Applies dark blue outline, blue fill.
• 'lineTrace' — Applies red outline, blue fill.
• 'unique' — Dark blue outline, white fill.
• 'user1', 'user2', 'user3', 'user4', 'user5' — Applies custom highlight: black outline, white fill by default (i.e., no highlight).

In addition, you can use these color schemes. The first word is the outline and the second is the fill color:

• 'orangeWhite'
• 'blackWhite'
• 'redWhite'
• 'blueWhite'
• 'greenWhite'

See Also
find_system | rtwtrace

Introduced before R2006a
isLoaded

(Not recommended) Determine if project is loaded

**Note** simulinkproject is not recommended. Use currentProject or openProject instead. For more information, see “Compatibility Considerations”.

**Syntax**

```matlab
loaded = isLoaded(proj)
```

**Description**

`loaded = isLoaded(proj)` returns whether the project referenced by the project object `proj` is loaded.

**Examples**

**Find Out if Project Is Loaded**

Open the airframe project and create a project object.

```matlab
sldemo_slproject_airframe;
proj = simulinkproject;
```

Find out if the project is still loaded.

```matlab
loaded = isLoaded(proj)
```
loaded =
    1

**Input Arguments**

proj — Project
project

Project, specified as a project object. Use simulinkproject to create a project object to manipulate a Project at the command line.

**Output Arguments**

loaded — Loaded status
1 | 0

Project loaded status, returned as true (1) if the project is loaded.

Data Types: logical

**Compatibility Considerations**

**Simulink project API is not recommended**

*Not recommended starting in R2019a*

Starting in R2019a, instead of simulinkproject and related functions, use the currentProject or openProject functions in MATLAB. The Simulink project API will continue to be supported, but, after R2019a, new features might be available only if you use the new MATLAB project API. There are no plans to remove the Simulink project API at this time.

The new MATLAB project API is part of the R2019a functionality enabling you to use projects in MATLAB, with or without Simulink. You can now share projects with users who do not have Simulink.
See Also

Functions
reload | simulinkproject

Introduced in R2013a
**legacy_code**

Use Legacy Code Tool

**Syntax**

```matlab
legacy_code('help')
specs = legacy_code('initialize')
legacy_code('sfcn_cmex_generate', specs)
legacy_code('compile', specs, compilerOptions)
legacy_code('generate_for_sim', specs, modelname)
legacy_code('sblock_generate', specs, modelname)
legacy_code('sfcn_tlc_generate', specs)
legacy_code('sfcn_makecfg_generate', specs)
legacy_code('rtwmakecfg_generate', specs)
legacy_code('backward_compatibility')
```

**Description**

The `legacy_code` function creates a MATLAB structure for registering the specification for existing C or C++ code and the S-function being generated. In addition, the function can generate, compile and link, and create a masked block for the specified S-function. Other options include generating

- A TLC file for simulation in Accelerator mode or code generation
- An `rtwmakecfg.m` file that you can customize to specify dependent source and header files that reside in a different directory than that of the generated S-function

`legacy_code('help')` displays instructions for using Legacy Code Tool.

`specs = legacy_code('initialize')` initializes the Legacy Code Tool data structure, `specs`, which registers characteristics of existing C or C++ code and properties of the S-function that the Legacy Code Tool generates.

`legacy_code('sfcn_cmex_generate', specs)` generates an S-function source file as specified by the Legacy Code Tool data structure, `specs`. 
legacy_code('compile', specs, compilerOptions) compiles and links the S-function generated by the Legacy Code Tool based on the data structure, specs, and compiler options that you might specify. The compiler options must be supported by the mex (MATLAB) function.

legacy_code('generate_for_sim', specs, modelname) generates, compiles, and links the S-function in a single step. If the Options.useTlcWithAccel field of the Legacy Code Tool data structure is set to logical 1 (true), the function also generates a TLC file for accelerated simulations.

legacy_code('slblock_generate', specs, modelname) generates a masked S-Function block for the S-function generated by the Legacy Code Tool based on the data structure, specs. The block appears in the Simulink model specified by modelname. If you omit modelname, the block appears in an empty model editor window.

legacy_code('sfcn_tlc_generate', specs) generates a TLC file for the S-function generated by the Legacy Code Tool based on the data structure, specs. This option is relevant if you want to:

- Force Accelerator mode in Simulink software to use the TLC inlining code of the generated S-function. See the description of the ssSetOptions SimStruct function and SS_OPTION_USE_TLC_WITH_ACCELERATOR S-function option for more information.
- Use Simulink Coder software to generate code from your Simulink model. For more information, see “Import Calls to External Code into Generated Code with Legacy Code Tool” (Simulink Coder).

legacy_code('sfcn_makecfg_generate', specs) generates an sFunction_makecfg.m file for the S-function generated by the Legacy Code Tool based on the data structure, specs. This option is relevant only if you use Simulink Coder software to generate code from your Simulink model. For more information, see “Use makecfg to Customize Generated Makefiles for S-Functions” (Simulink Coder) and “Import Calls to External Code into Generated Code with Legacy Code Tool” (Simulink Coder).

legacy_code('rtwmakecfg_generate', specs) generates an rtwmakecfg.m file for the S-function generated by the Legacy Code Tool based on the data structure, specs. This option is relevant only if you use Simulink Coder software to generate code from your Simulink model. For more information, see “Use rtwmakecfg.m API to Customize Generated Makefiles” (Simulink Coder) and “Import Calls to External Code into Generated Code with Legacy Code Tool” (Simulink Coder).
legacy_code('backward_compatibility') automatically updates syntax for using Legacy Code Tool to the supported syntax described in this reference page and in “Integrate C Functions Using Legacy Code Tool”.

**Input Arguments**

`sargs`

A structure with the following fields:

**Name the S-function**

`SFunctionName` (Required) — A character vector specifying a name for the S-function to be generated by the Legacy Code Tool.

**Define Legacy Code Tool Function Specifications**

- **InitializeConditionsFcnSpec** — A nonempty character vector specifying a function that the S-function calls to initialize and reset states. You must declare this function by using tokens that Simulink software can interpret as explained in “Declaring Legacy Code Tool Function Specifications”.
- **OutputFcnSpec** — A nonempty character vector specifying the function that the S-function calls at each time step. You must declare this function by using tokens that Simulink software can interpret as explained in “Declaring Legacy Code Tool Function Specifications”.
- **StartFcnSpec** — A character vector specifying the function that the S-function calls when it begins execution. This function can access S-function parameter arguments and work data. You must declare this function by using tokens that Simulink software can interpret as explained in “Declaring Legacy Code Tool Function Specifications”.
- **TerminateFcnSpec** — A character vector specifying the function that the S-function calls when it terminates execution. This function can access S-function parameter arguments and work data. You must declare this function by using tokens that Simulink software can interpret as explained in “Declaring Legacy Code Tool Function Specifications”.

**Define Compilation Resources**

- **HeaderFiles** — A cell array of character vectors specifying the file names of header files required for compilation.
• **SourceFiles** — A cell array of character vectors specifying source files required for compilation. You can specify the source files using absolute or relative path names.

• **HostLibFiles** — A cell array of character vectors specifying library files required for host compilation. You can specify the library files using absolute or relative path names.

• **TargetLibFiles** — A cell array of character vectors specifying library files required for target (that is, standalone) compilation. You can specify the library files using absolute or relative path names.

• **IncPaths** — A cell array of character vectors specifying directories containing header files. You can specify the directories using absolute or relative path names.

• **SrcPaths** — A cell array of character vectors specifying directories containing source files. You can specify the directories using absolute or relative path names.

• **LibPaths** — A cell array of character vectors specifying directories containing host and target library files. You can specify the directories using absolute or relative path names.

**Specify a Sample Time**

**SampleTime** — One of the following:

• `'inherited'` (default) — Sample time is inherited from the source block.

• `'parameterized'` — Sample time is represented as a tunable parameter. Generated code can access the parameter by calling MEX API functions, such as `mxGetPr` or `mxGetData`.

• **Fixed** — Sample time that you explicitly specify. For information on how to specify sample time, see “Specify Sample Time”.

If you specify this field, you must specify it last.

**Define S-Function Options**

**Options** — A structure that controls S-function options. The structure's fields include:

• **isMacro** — A logical value specifying whether the legacy code is a C macro. By default, the value is false (0).

• **isVolatile** — A logical value specifying the setting of the S-function `SS_OPTION_NONVOLATILE` option. By default, the value is true (1).
• **canBeCalledConditionally** — A logical value specifying the setting of the S-function SS_OPTION_CAN_BE_CALLED_CONDITIONALLY option. By default, the value is true (1).

• **useTlcWithAccel** — A logical value specifying the setting of the S-function SS_OPTION_USE_TLC_WITH_ACCELERATOR option. By default, the value is true (1).

• **language** — A character vector specifying either 'C' or 'C++' as the target language of the S-function that Legacy Code Tool will produce. By default, the value is 'C'.

**Note** The Legacy Code Tool can interface with C++ functions, but not C++ objects. For a work around, see “Legacy Code Tool Limitations” in the Simulink documentation.

• **singleCPPMexFile** — A logical value that, if true, specifies that generated code:
  
  • Requires you to generate and manage an inlined S-function as only one file (.cpp) instead of two (.c and .tlc).
  • Maintains model code style (level of parentheses usage and preservation of operand order in expressions and condition expressions in if statements) as specified by model configuration parameters.

By default, the value is false.

**Limitations** You cannot set the singleCPPMexFile field to true if

• Options.language='C++'

• You use one of the following Simulink objects with the IsAlias property set to true:
  
  • Simulink.Bus
  • Simulink.AliasType
  • Simulink.NumericType

• The Legacy Code Tool function specification includes a void* or void** to represent scalar work data for a state argument

• HeaderFiles field of the Legacy Code Tool structure specifies multiple header files
• **supportsMultipleExecInstances**— A logical value specifying whether to include a call to the `ssSupportsMultipleExecInstances` function. By default, the value is `false (0)`.

• **convertNDArrayToRowMajor**— A logical value specifying the automatic conversion of a matrix between a column-major format and a row-major format. The column-major format is used by MATLAB, Simulink, and the generated code. The row-major format is used by C. By default, the value is `false (0)`. If you currently specify the previous version of the option, `convert2DMatrixToRowMajor`, the function automatically specifies the new `convertNDArrayToRowMajor` option.

**Note** This option does not support a 2-D matrix of complex data.

• **supportCoverage**— A logical value specifying whether the generated S-function must be compatible with Model Coverage. By default, the value is `false (0)`.

• **supportCoverageAndDesignVerifier**— A logical value specifying whether the generated S-function must be compatible with Model Coverage and Simulink Design Verifier™. By default, the value is `false (0)`.

• **outputsConditionallyWritten**— A logical value specifying whether the legacy code conditionally writes the output ports. If `true`, the generated S-function specifies that the memory associated with each output port cannot be overwritten and is global (`SS_NOT_REUSABLE_AND_GLOBAL`). If `false`, the memory associated with each output port is reusable and is local (`SS_REUSABLE_AND_LOCAL`). By default, the value is `false (0)`. For more information, see `ssSetOutputPortOptimOpts`.

**modelname**

The name of a Simulink model into which Legacy Code Tool is to insert the masked S-function block generated when you specify `legacy_code` with the action character vector `'slblock_generate'`. If you omit this argument, the block appears in an empty model editor window.

**See Also**

**Topics**

“Integrate C Functions Using Legacy Code Tool”
“Import Calls to External Code into Generated Code with Legacy Code Tool” (Simulink Coder)

**Introduced in R2006b**
libinfo

Get information about library blocks referenced by model

Syntax

libdata = libinfo('system')
libdata = libinfo('system', constraint1, value1, ...)

Description

libdata = libinfo('system') returns information about library blocks referenced by system and all the systems underneath it.

libdata = libinfo('system', constraint1, value1, ...) restricts the search as indicated by the search constraint(s) c1, v1, ...

Input Arguments

system

The system to search recursively for library blocks.

constraint1, value1, ...

One or more pairs, each consisting of a search constraint followed by a constraint value. You can specify any of the search constraints that you can use with find_system.

Output Arguments

libdata

An array of structures that describes each library block referenced by system. Each structure has the following fields:
Block
Library
ReferenceBlock
LinkStatus

Path of the link to the library block
Name of the library containing the referenced block
Path of the library block
Value of the LinkStatus parameter for the link to the library block

See Also
find_system

Topics
“Create a Custom Library”

Introduced before R2006a
linmod

Extract continuous-time linear state-space model around operating point

Syntax

argout = linmod('sys');
argout = linmod('sys', x, u);
argout = linmod('sys', x, u, para);
argout = linmod('sys', x, u, 'v5');
argout = linmod('sys', x, u, para, 'v5');
argout = linmod('sys', x, u, para, xpert, upert, 'v5');

Arguments

sys
State (x) and the input (u) vectors. If specified, they set the operating point at which the linear model is extracted. When a model has model references using the Model block, you must use the Simulink structure format to specify x. To extract the x structure from the model, use the following command:

x = Simulink.BlockDiagram.getInitialState('sys');

You can then change the operating point values within this structure by editing x.signals.values.

If the state contains different data types (for example, 'double' and 'uint8'), then you cannot use a vector to specify this state. You must use a structure instead. In addition, you can only specify the state as a vector if the state data type is 'double'.

x and u

Name of the Simulink system from which the linear model is extracted.

Ts
Sample time of the discrete-time linearized model

'v5'
An optional argument that invokes the perturbation algorithm created prior to MATLAB 5.3. Invoking this optional argument is equivalent to calling linmodv5.
para

A three-element vector of optional arguments:

- **para(1)** — Perturbation value of delta, the value used to perform the perturbation of the states and the inputs of the model. This is valid for linearizations using the 'v5' flag. The default value is 1e-05.

- **para(2)** — Linearization time. For blocks that are functions of time, you can set this parameter with a nonnegative value that gives the time (t) at which Simulink evaluates the blocks when linearizing a model. The default value is 0.

- **para(3)** — Set para(3)=1 to remove extra states associated with blocks that have no path from input to output. The default value is 0.

**xpert** and **upert**

The perturbation values used to perform the perturbation of all the states and inputs of the model. The default values are

\[
\begin{align*}
\text{xpert} &= \text{para}(1) + 1e-3\times\text{para}(1)\times\text{abs}(x) \\
\text{upert} &= \text{para}(1) + 1e-3\times\text{para}(1)\times\text{abs}(u)
\end{align*}
\]

When a model has model references using the Model block, you must use the Simulink structure format to specify xpert. To extract the xpert structure, use the following command:

\[
\text{xpert} = \text{Simulink.BlockDiagram.getInitialState('sys');}
\]

You can then change the perturbation values within this structure by editing xpert.signals.values.

The perturbation input arguments are only available when invoking the perturbation algorithm created prior to MATLAB 5.3, either by calling linmodv5 or specifying the 'v5' input argument to linmod.
linmod, dlinmod, and linmod2 return state-space representations if you specify the output (left-hand) side of the equation as follows:

• \([A,B,C,D] = \text{linmod}('sys', x, u)\) obtains the linearized model of \(sys\) around an operating point with the specified state variables \(x\) and the input \(u\). If you omit \(x\) and \(u\), the default values are zero.

linmod and dlinmod both also return a transfer function and MATLAB data structure representations of the linearized system, depending on how you specify the output (left-hand) side of the equation. Using \text{linmod} as an example:

• \([\text{num, den}] = \text{linmod}('sys', x, u)\) returns the linearized model in transfer function form.
• \(\text{sys\_struc} = \text{linmod}('sys', x, u)\) returns a structure that contains the linearized model, including state names, input and output names, and information about the operating point.

Description

\text{linmod} compute a linear state-space model by linearizing each block in a model individually.

\text{linmod} obtains linear models from systems of ordinary differential equations described as Simulink models. Inputs and outputs are denoted in Simulink block diagrams using Inport and Outport blocks.

The default algorithm uses preprogrammed analytic block Jacobians for most blocks which should result in more accurate linearization than numerical perturbation of block inputs and states. A list of blocks that have preprogrammed analytic Jacobians is available in the Simulink Control Design documentation along with a discussion of the block-by-block analytic algorithm for linearization.

The default algorithm also allows for special treatment of problematic blocks such as the Transport Delay and the Quantizer. See the mask dialog of these blocks for more information and options.
Notes

By default, the system time is set to zero. For systems that are dependent on time, you can set the variable `para` to a two-element vector, where the second element is used to set the value of `t` at which to obtain the linear model.

The ordering of the states from the nonlinear model to the linear model is maintained. For Simulink systems, a character vector variable that contains the block name associated with each state can be obtained using

```
[sizes,x0,xstring] = sys
```

where `xstring` is a vector of strings whose `i`th row is the block name associated with the `i`th state. Inputs and outputs are numbered sequentially on the diagram.

For single-input multi-output systems, you can convert to transfer function form using the routine `ss2tf` or to zero-pole form using `ss2zp`. You can also convert the linearized models to LTI objects using `ss`. This function produces an LTI object in state-space form that can be further converted to transfer function or zero-pole-gain form using `tf` or `zpk`.

The default algorithms in `linmod` handle Transport Delay blocks by replacing the linearization of the blocks with a Padé approximation. For the 'v5' algorithm, linearization of a model that contains Derivative or Transport Delay blocks can be troublesome. For more information, see “Linearizing Models”.

See Also

dlinmod | linmod2 | linmodv5

Introduced in R2007a
linmod2

Extract continuous-time linear state-space model around operating point

Syntax

argout = linmod2('sys', x, u);
argout = linmod2('sys', x, u, para);

Arguments

sys  Name of the Simulink system from which the linear model is extracted.
x, u State (x) and the input (u) vectors. If specified, they set the operating point at which the linear model is extracted. When a model has model references using the Model block, you must use the Simulink structure format to specify x. To extract the x structure from the model, use the following command:

x = Simulink.BlockDiagram.getInitialState('sys');

You can then change the operating point values within this structure by editing x.signals.values.

If the state contains different data types (for example, 'double' and 'uint8'), then you cannot use a vector to specify this state. You must use a structure instead. In addition, you can only specify the state as a vector if the state data type is 'double'.

para A three-element vector of optional arguments:

- para(1) — Perturbation value of delta, the value used to perform the perturbation of the states and the inputs of the model. This is valid for linearizations using the 'v5' flag. The default value is 1e-05.
- para(2) — Linearization time. For blocks that are functions of time, you can set this parameter with a nonnegative value that gives the time (t) at which Simulink evaluates the blocks when linearizing a model. The default value is 0.
- para(3) — Set para(3)=1 to remove extra states associated with blocks that have no path from input to output. The default value is 0.

argout linmod, dlinmod, and linmod2 return state-space representations if you specify the output (left-hand) side of the equation as follows:

- \([A,B,C,D] = \text{linmod}('sys', x, u)\) obtains the linearized model of \(sys\) around an operating point with the specified state variables \(x\) and the input \(u\). If you omit \(x\) and \(u\), the default values are zero.

linmod and dlinmod both also return a transfer function and MATLAB data structure representations of the linearized system, depending on how you specify the output (left-hand) side of the equation. Using \text{linmod} as an example:

- \([\text{num}, \text{den}] = \text{linmod}('sys', x, u)\) returns the linearized model in transfer function form.
- \(\text{sys\_struc} = \text{linmod}('sys', x, u)\) returns a structure that contains the linearized model, including state names, input and output names, and information about the operating point.

Description

\text{linmod2} computes a linear state-space model by perturbing the model inputs and model states, and uses an advanced algorithm to reduce truncation error.
linmod2 obtains linear models from systems of ordinary differential equations described as Simulink models. Inputs and outputs are denoted in Simulink block diagrams using Inport and Outport blocks.

**Notes**

By default, the system time is set to zero. For systems that are dependent on time, you can set the variable `para` to a two-element vector, where the second element is used to set the value of `t` at which to obtain the linear model.

The ordering of the states from the nonlinear model to the linear model is maintained. For Simulink systems, a character vector variable that contains the block name associated with each state can be obtained using

```
[sizes,x0,xstring] = sys
```

where `xstring` is a vector of strings whose `i`th row is the block name associated with the `i`th state. Inputs and outputs are numbered sequentially on the diagram.

For single-input multi-output systems, you can convert to transfer function form using the routine `ss2tf` or to zero-pole form using `ss2zp`. You can also convert the linearized models to LTI objects using `ss`. This function produces an LTI object in state-space form that can be further converted to transfer function or zero-pole-gain form using `tf` or `zpk`.

The default algorithms in `linmod` and `dlinmod` handle Transport Delay blocks by replacing the linearization of the blocks with a Pade approximation. For more information, see “Linearizing Models”.

**See Also**

`dlinmod` | `linmod` | `linmodv5`

**Introduced in R2007a**
linmodv5

Extract continuous-time linear state-space model around operating point

Syntax

argout = linmodv5('sys');
argout = linmodv5('sys',x,u);
argout = linmodv5('sys', x, u, para);
argout = linmodv5('sys', x, u, para, xpert, upert);

Arguments

sys Name of the Simulink system from which the linear model is extracted.

x, u State (x) and the input (u) vectors. If specified, they set the operating point at which the linear model is extracted. When a model has model references using the Model block, you must use the Simulink structure format to specify x. To extract the x structure from the model, use the following command:

x = Simulink.BlockDiagram.getInitialState('sys');

You can then change the operating point values within this structure by editing x.signals.values.

If the state contains different data types (for example, 'double' and 'uint8'), then you cannot use a vector to specify this state. You must use a structure instead. In addition, you can only specify the state as a vector if the state data type is 'double'.

2-416
**para**  
A three-element vector of optional arguments:

- **para(1)** — Perturbation value of delta, the value used to perform the perturbation of the states and the inputs of the model. This is valid for linearizations using the 'v5' flag. The default value is 1e-05.

- **para(2)** — Linearization time. For blocks that are functions of time, you can set this parameter with a nonnegative value that gives the time \(t\) at which Simulink evaluates the blocks when linearizing a model. The default value is 0.

- **para(3)** — Set para(3)=1 to remove extra states associated with blocks that have no path from input to output. The default value is 0.

**xpert, upert**  
The perturbation values used to perform the perturbation of all the states and inputs of the model. The default values are

\[
xpert = \text{para}(1) + 1e-3\times\text{para}(1)\times\text{abs}(x)
\]
\[
upert = \text{para}(1) + 1e-3\times\text{para}(1)\times\text{abs}(u)
\]

When a model has model references using the Model block, you must use the Simulink structure format to specify xpert. To extract the xpert structure, use the following command:

\[
xpert = \text{Simulink.BlockDiagram.getInitialState('sys')};
\]

You can then change the perturbation values within this structure by editing `xpert.signals.values`.

The perturbation input arguments are only available when invoking the perturbation algorithm created prior to MATLAB 5.3, either by calling `linmodv5` or specifying the 'v5' input argument to `linmod`.  

2-417
linmod, dlinmod, and linmod2 return state-space representations if you specify the output (left-hand) side of the equation as follows:

- \([A, B, C, D] = \text{linmod}('sys', x, u)\) obtains the linearized model of \(sys\) around an operating point with the specified state variables \(x\) and the input \(u\). If you omit \(x\) and \(u\), the default values are zero.

linmod and dlinmod both also return a transfer function and MATLAB data structure representations of the linearized system, depending on how you specify the output (left-hand) side of the equation. Using linmod as an example:

- \([\text{num}, \text{den}] = \text{linmod}('sys', x, u)\) returns the linearized model in transfer function form.
- \(\text{sys}_\text{struc} = \text{linmod}('sys', x, u)\) returns a structure that contains the linearized model, including state names, input and output names, and information about the operating point.

### Description

linmodv5 computes a linear state space model using the full model perturbation algorithm created prior to MATLAB 5.3.

linmodv5 obtains linear models from systems of ordinary differential equations described as Simulink models. Inputs and outputs are denoted in Simulink block diagrams using Inport and Outport blocks.

### Notes

By default, the system time is set to zero. For systems that are dependent on time, you can set the variable \(\text{para}\) to a two-element vector, where the second element is used to set the value of \(t\) at which to obtain the linear model.

The ordering of the states from the nonlinear model to the linear model is maintained. For Simulink systems, a character vector variable that contains the block name associated with each state can be obtained using
[sizes,x0,xstring] = sys

where xstring is a vector of strings whose ith row is the block name associated with the ith state. Inputs and outputs are numbered sequentially on the diagram.

For single-input multi-output systems, you can convert to transfer function form using the routine ss2tf or to zero-pole form using ss2zp. You can also convert the linearized models to LTI objects using ss. This function produces an LTI object in state-space form that can be further converted to transfer function or zero-pole-gain form using tf or zpk.

The default algorithms in linmod and dlinmod handle Transport Delay blocks by replacing the linearization of the blocks with a Pade approximation. For the 'v5' algorithm, linearization of a model that contains Derivative or Transport Delay blocks can be troublesome. For more information, see “Linearizing Models”.

See Also
dlinmod | linmod | linmod2

Introduced in R2011b
load_system

Load Simulink model into memory

Syntax

handle = load_system(sys)

Description

handle = load_system(sys) loads the model sys into memory without opening the model in the Simulink Editor. After you load a model into memory, you can work with it using Simulink API commands. Save changes to the model using save_system.

Examples

Load Model into Memory

Load the model vdp into memory and return the model handle.

h = load_system("vdp")

h =

172.0004

Input Arguments

sys — Model to load into memory
character vector | string scalar

Model to load into memory, specified as a character vector or string scalar.

Example: "vdp"
Output Arguments

`handle` — Handle of loaded model

`handle`

Handle of loaded model.

See Also

`close_system` | `open_system` | `save_system`

Introduced before R2006a
loadIntoMemory

Load logged data into memory

Syntax

logs = loadIntoMemory(logs)

Description

logs = loadIntoMemory(logs) loads the data in logs into memory. Data is logged to a repository and brought into memory on an as-needed basis. When you want to work with all elements of a large set of logged data, use loadIntoMemory to bring all of the elements into memory at once. Loading all the data at once, rather than element by element, is much faster.

Examples

Load Logged Data into Memory

This example shows how to load a set of logged data into memory all at once, rather than element by element.

% Simulate model to generate logged data
sim('sldemo_fuelsys')

The simulation logs all of the instrumented signals in the model to the Simulink.SimulationData.Dataset object sldemo_fuelsys_output. At the end of simulation, the signal data remains in the repository until used in the MATLAB workspace. When you work with small sets of data or only postprocess a subset, leaving signals in the repository improves performance. But when you have a large set of data and need to postprocess all of the signals, you should bring them all into memory at once.

% Load all logged signals into memory
loadIntoMemory(sldemo_fuelsys_output);
All of the data in sldemo_fuelsys_output is now available for efficient postprocessing.

**Input Arguments**

logs — Data to load into memory
'Simulink.SimulationData.Dataset' | 'Simulink.SimulationOutput'

Data to load into memory. The loadIntoMemory function can load Simulink.SimulationData.Dataset and Simulink.SimulationOutput data.

Example: logsout

**Output Arguments**

logs — Data
'Simulink.SimulationData.Dataset' | 'Simulink.SimulationOutput'

Data loaded into memory.

**See Also**

Simulink.SimulationData.Dataset | Simulink.SimulationOutput

Introduced in R2017b
model

Execute particular phase of simulation of model

Syntax

[sys,x0,str,ts] = model([],[],[],'sizes');
[sys,x0,str,ts] = model([],[],[],'compile');
outputs = model(t,x,u,'outputs');
derivs = model(t,x,u,'derivs');
dstates = model(t,x,u,'update');
model([],[],[],'term');

Description

The `model` command executes a specific phase of the simulation of a Simulink model whose name is `model`. The command’s last argument (flag) specifies the phase of the simulation to be executed. See “Simulation Phases in Dynamic Systems” for a description of the steps that Simulink software uses to simulate a model.

This command ignores the effects of state transitions and conditional execution. Therefore, it is not suitable for models which have such logic. Use this command for models which can be represented as simple dynamic systems. Such systems should meet these requirements.

- All states in the model must be built-in non-bus data types. For a discussion on built-in data types, see “About Data Types in Simulink”.
- If you are using vector format to specify the state, this command can access only non-complex states of `double` data type.
- There is minimal amount of state logic (Stateflow, conditionally executed subsystems etc.)
- The models are not mixed-domain models. That is, most blocks in the model are built-in Simulink blocks and do not include user-written S-functions or blocks from other Sim* products.
For models which do not comply with these requirements, using this command can cause Simulink to produce results which can only be interpreted by further analyzing and simplifying the model.

**Note** The state variable \( x \) can be represented in structure as well as vector formats. The variable follows the limitations of the format in which it is specified.

This command is also not intended to be used to run a model step-by-step, for example, to debug a model. Use the Simulink debugger if you need to examine intermediate results to debug a model.

**Arguments**

<table>
<thead>
<tr>
<th><strong>sys</strong></th>
<th>Vector of model size data:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>• ( \text{sys}(1) ) = number of continuous states</td>
</tr>
<tr>
<td></td>
<td>• ( \text{sys}(2) ) = number of discrete states</td>
</tr>
<tr>
<td></td>
<td>• ( \text{sys}(3) ) = number of outputs</td>
</tr>
<tr>
<td></td>
<td>• ( \text{sys}(4) ) = number of inputs</td>
</tr>
<tr>
<td></td>
<td>• ( \text{sys}(5) ) = reserved</td>
</tr>
<tr>
<td></td>
<td>• ( \text{sys}(6) ) = direct-feedthrough flag (1 = yes, 0 = no)</td>
</tr>
<tr>
<td></td>
<td>• ( \text{sys}(7) ) = number of Continuous, Discrete, Fixed in minor step, and Controllable sample times (= number of rows in ( ts ))</td>
</tr>
</tbody>
</table>

| **x0** | Vector containing the initial conditions of the system's states |
| **str** | Vector of names of the blocks associated with the model's states. The state names and initial conditions appear in the same order in \( \text{str} \) and \( x0 \), respectively. |

<p>| <strong>ts</strong> | An ( m )-by-2 matrix containing the sample time (period, offset) information of the Continuous, Discrete, Fixed in minor step, and Controllable sample times in the model. For more information on the sample times in Simulink, see “Types of Sample Time”. |</p>
<table>
<thead>
<tr>
<th>outputs</th>
<th>Outputs of the model at time step t.</th>
</tr>
</thead>
<tbody>
<tr>
<td>derivs</td>
<td>Derivatives of the continuous states of the model at time t.</td>
</tr>
<tr>
<td>dstates</td>
<td>States of the model at time t returned as either a structure or an array. Simulink returns a structure when the model has states and x is either empty ([ ]) or in structure format. Otherwise, Simulink returns an array.</td>
</tr>
<tr>
<td></td>
<td>• If the return type is a vector or array, Simulink returns real double discrete states only.</td>
</tr>
<tr>
<td></td>
<td>• If the return type is a structure, Simulink returns a structure that contains both continuous and discrete states of built-in types only. Non-built-in types are omitted.</td>
</tr>
<tr>
<td>t</td>
<td>Time step, specified as real double in scalar format.</td>
</tr>
<tr>
<td>x</td>
<td>State vector, specified as real double in structure or vector format.</td>
</tr>
<tr>
<td>u</td>
<td>Inputs, specified as real double in vector format.</td>
</tr>
</tbody>
</table>
flag

<table>
<thead>
<tr>
<th>Flag</th>
<th>Specification of the simulation phase to be executed:</th>
</tr>
</thead>
<tbody>
<tr>
<td>'sizes'</td>
<td>executes the size computation phase of the simulation. This phase determines the sizes of the model's inputs, outputs, state vector, etc.</td>
</tr>
<tr>
<td>'compile'</td>
<td>executes the compilation phase of the simulation. The compilation phase propagates signal and sample time attributes.</td>
</tr>
<tr>
<td>'update'</td>
<td>computes the next values of the model's discrete states.</td>
</tr>
<tr>
<td>'outputs'</td>
<td>computes the outputs of the model's blocks at time ( t ).</td>
</tr>
<tr>
<td>'derivs'</td>
<td>computes the derivatives of the model's continuous states at time step ( t ).</td>
</tr>
<tr>
<td>'term'</td>
<td>causes Simulink software to terminate simulation of the model.</td>
</tr>
</tbody>
</table>

**Note** The output, update, and derivs flags are valid only for single-tasking models. For more information on single-tasking and multi-tasking, see “Tasking Modes” (Simulink Coder).

### Examples

The following command executes the compilation phase of the vdp model that comes with Simulink software.

```matlab
vdp([], [], [], 'compile')
```

The following command terminates the simulation initiated in the previous example.

```matlab
vdp([], [], [], 'term')
```

**Note** Simulink does not let you close a model while it is compiling or simulating. For all phases except the 'sizes' phase, before closing the model, you must invoke the model command with the 'term' argument.
See Also

sim

Introduced in R2007a
modeladvisor

Open Model Advisor

Syntax

modeladvisor(model)

Description

modeladvisor(model) opens the Model Advisor for the model or subsystem specified by model. If the specified model or subsystem is not open, this command opens it.

Examples

Open Model Advisor for model

Open the Model Advisor for vdp example model:

modeladvisor('vdp')

Open Model Advisor for subsystem

Open the Model Advisor for the Aircraft Dynamics Model subsystem of the f14 example model:

modeladvisor('f14/Aircraft Dynamics Model')

Open Model Advisor for currently selected model

Open the Model Advisor on the currently selected model:
modeladvisor(bdroot)

Open Model Advisor for currently selected subsystem
Open the Model Advisor on the currently selected subsystem:
modeladvisor(gcs)

Input Arguments

model — Model or subsystem name
character vector
Model or subsystem name or handle, specified as a character vector.
Data Types: char

See Also
“Run Model Advisor Checks”

Introduced before R2006a
new_system

Create Simulink model or library in memory

Syntax

h = new_system

h = new_system(name)
h = new_system(name,'FromTemplate',template)
h = new_system(name,'FromFile',file)
h = new_system(___,'ErrorIfShadowed')

h = new_system(name,'Model')
h = new_system(name,'Model',subsys)
h = new_system(name,'Subsystem')
h = new_system(name,'Library')
h = new_system(___,'ErrorIfShadowed')

Description

h = new_system creates a model named untitled (and then untitled1, untitled2, and so on) based on your default model template and returns the new model’s numeric handle. Select your default model template on the Simulink start page or by using the Simulink.defaultModelTemplate function.

The new_system function does not open the new model. This function creates the model in memory. To save the model, use save_system, or open the model with open_system and then save it using the Simulink Editor.

h = new_system(name) creates a model based on your default model template and gives the new model the specified name. This function returns the new model’s numeric handle. Select your default model template on the Simulink start page or by using the Simulink.defaultModelTemplate function.

If name is empty, the function creates a model named untitled, untitled1, untitled2, and so on.
The `new_system` function does not open the new model. This function creates the model in memory. To save the model, use `save_system`, or open the model with `open_system` and then save it using the Simulink Editor.

```matlab
h = new_system(name,'FromTemplate',template) creates the model based on the specified template.
```

```matlab
h = new_system(name,'FromFile',file) creates the model based on the specified model or template.
```

```matlab
h = new_system(___,'ErrorIfShadowed') creates the model or returns an error if another model, MATLAB file, or variable with the same name is on the MATLAB path or in the workspace. It uses any of the input arguments in the previous syntaxes.
```

```matlab
h = new_system(name,'Model') creates an empty model based on the Simulink default model and returns the new model's numeric handle. The Simulink default model is also known as the root block diagram and has the numeric handle 0. If name is empty, the function creates a model or library named `untitled`, `untitled1`, `untitled2`, and so on.
```

The `new_system` function does not open the new model. This function creates the model in memory. To save the model, use `save_system`, or open the model with `open_system` and then save it using the Simulink Editor.

```matlab
h = new_system(name,'Model',subsys) creates a model based on the subsystem `subsys` in a currently loaded model.
```

```matlab
h = new_system(name,'Subsystem') creates an empty subsystem file that has the specified name.
```

```matlab
h = new_system(name,'Library') creates an empty library that has the specified name and returns a numeric handle.
```

```matlab
h = new_system(___,'ErrorIfShadowed') returns an error if another model, MATLAB file, or variable with the same name is on the MATLAB path or in the workspace. This syntax uses any of the input arguments in the previous syntaxes.
```

**Examples**
**Create a Model Based on Default Template**

Create a model in memory called untitled.

```matlab
h = new_system;
```

You can use `get_param` to get the name.

```matlab
get_param(h,'Name')
open_system(h)
open_system('untitled')
open_system(get_param(h,'Name'))
```

```matlab
ans =
untitled
```

Use the name, the handle, or `get_param` command as input to `open_system`. You can use any of these commands:

**Create a Model Based on Named Template**

Before you use this syntax, create a template. In the Simulink Editor, create the model you want to use as the template, then in the **Simulation** tab, select **Save > Template**. For this example, name the template `mytemplate`.

By default, the template is on the MATLAB path, so if you change location, add the folder to the MATLAB path.

Create a model named `templateModel` based on the template `mytemplate`.

```matlab
h = new_system('templateModel','FromTemplate','mytemplate');
```

Invoking template `\myuserdir\Documents\MATLAB\mytemplate.sltx`

**Create a Model Based on Another Model**

Create a model named `mynewmodel` based on `myoldmodel`, which is in the current folder.
h = new_system('mynewmodel','FromFile','myoldmodel.slx');

**Create Model from Subsystem**

Load the model f14. Create a model based on the Controller subsystem.

load_system('f14');
new_system('mycontroller','Model','f14/Controller');
open_system('mycontroller');

**Create a Library**

Create a library in memory and then open it.
new_system('mylib','Library')
open_system('mylib')

Create a Subsystem File

Create a subsystem file in memory and then open it.

new_system('mysubsystem','Subsystem')
open_system('mysubsystem')

Ensure Model Name Is Unique

Create a variable with the name myvar.

myvar = 17

Try to create a model that uses the same name as the variable. When you use the 'ErrorIfShadowed' option, the new_system function returns an error.

new_system('myvar2','Model','ErrorIfShadowed')

The model 'myvar2' cannot be created because this name is shadowing another name on the MATLAB path or in the workspace. Choose another name, or do not use the option 'ErrorIfShadowed'.

Input Arguments

name — Name of new model or library
character vector

Name of new model or library, specified as a character vector that:

- Has 63 or fewer characters
- Is not a MATLAB keyword
- Is not 'simulink'
- Is unique among model names, variables, and MATLAB files on the MATLAB path and in the workspace

Example: 'mymodel', 'mylibrary'
subsys — Subsystem to base new model on
subsystem block path name

Subsystem to base the new model on, specified as the subsystem block path name in a currently open model.
Example: 'f14/Controller'

template — Name of template to base new model on
character vector

Name of the template to base the new model on, specified as a character vector of the name of a template on the MATLAB path. Create a template in the Simulink Editor. In the Simulation tab, select Save > Template.
Example: 'mytemplate','mytemplate.sltx'

file — Path name of model or template to base new model on
character vector

Path name of the model or template to base the new model on, specified as a character vector. You can use an .mdl, .slx, or .sltx file. Include the extension and use a full or relative path.
Example: 'Models/mymodel.slx','mytemplate.sltx','model.mdl'

See Also
Simulink.defaultModelTemplate | open_system | save_system

Introduced before R2006a
num2fixpt

Convert number to nearest value representable by specified fixed-point data type

Syntax

outValue = num2fixpt(OrigValue, FixPtDataType, FixPtScaling, 
                     RndMeth, DoSatur)

Description

num2fixpt(OrigValue, FixPtDataType, FixPtScaling, RndMeth, DoSatur) returns the result of converting OrigValue to the nearest value representable by the fixed-point data type FixPtDataType. Both OrigValue and outValue are of data type double. As illustrated in the example that follows, you can use num2fixpt to investigate quantization error that might result from converting a number to a fixed-point data type. The arguments of num2fixpt include:

OrigValue

Value to be converted to a fixed-point representation. Must be specified using a double data type.

FixPtDataType

The fixed-point data type used to convert OrigValue.

FixPtScaling

Scaling of the output in either Slope or [Slope Bias] format. If FixPtDataType does not specify a generalized fixed-point data type using the sfix or ufix command, FixPtScaling is ignored.

RndMeth

Rounding technique used if the fixed-point data type lacks the precision to represent OrigValue. If FixPtDataType specifies a floating-point data type using the float command, RndMeth is ignored. Valid values are Zero, Nearest, Ceiling, or Floor (the default).
DoSatur Indicates whether the output should be saturated to the minimum or maximum representable value upon underflow or overflow. If FixPtDataType specifies a floating-point data type using the float command, DoSatur is ignored. Valid values are on or off (the default).

Examples

Suppose you wish to investigate the quantization effect associated with representing the real-world value 9.875 as a signed, 8-bit fixed-point number. The command

num2fixpt(9.875, sfix(8), 2^-1)

ans =

9.50000000000000

reveals that a slope of $2^{-1}$ results in a quantization error of 0.375. The command

num2fixpt(9.875, sfix(8), 2^-2)

ans =

9.75000000000000

demonstrates that a slope of $2^{-2}$ reduces the quantization error to 0.125. But a slope of $2^{-3}$, as used in the command

num2fixpt(9.875, sfix(8), 2^-3)

ans =

9.87500000000000

eliminates the quantization error entirely.

See Also
fixptbestexp | fixptbestprec
Introduced before R2006a
**open_system**

Open model, library, subsystem, or block dialog box

**Syntax**

open_system(obj)

double-clicking the model or library in the Current Folder Browser, or the subsystem or block in the Simulink Editor.

A model or library opens in a new window. For a subsystem or block within a model, the behavior depends on the type of block and its properties.

- Any OpenFcn callback parameter is evaluated.
- If there is no OpenFcn callback, and a mask is defined, the mask parameter dialog box opens.
- Without an OpenFcn callback or a mask parameter, Simulink opens the object.
    - A referenced model opens in a new window.
    - A subsystem opens in a new tab in the same window.
    - For blocks, the parameters dialog box for the block opens.
To open a specific subsystem or block, you must load the model or library containing it. Otherwise Simulink returns an error.

You can override the default behavior by supplying a second input argument.

`open_system(sys,'loadonly')` loads the specified model or library without opening the Simulink Editor. This is equivalent to using `load_system`.

`open_system(sbsys,'window')` opens the subsystem `sbsys` in a new Simulink Editor window. Before opening a specific subsystem or block, load the model or library containing it. Otherwise Simulink returns an error.

`open_system(sbsys,'tab')` opens the subsystem in a new Simulink Editor tab in the same window. Before opening a specific subsystem or block, load the model or library containing it. Otherwise Simulink returns an error.

`open_system(blk,'mask')` opens the mask dialog box of the block or subsystem specified by `blk`. Load the model or library containing `blk` before opening it.

`open_system(blk,'force')` looks under the mask of a masked block or subsystem. It opens the dialog box of the block under the mask or opens a masked subsystems in a new Simulink Editor tab. This is equivalent to the Look Under Mask menu item. Before opening a specific subsystem or block, load the model or library containing it. Otherwise Simulink returns an error.

`open_system(blk,'parameter')` opens the block parameter dialog box.

`open_system(blk,'OpenFcn')` runs the block callback `OpenFcn`.

**Examples**

**Open a Model**

Open the `f14` model.

`open_system('f14')`
Load a Model Without Opening it

Load the f14 model.

open_system('f14','loadonly')

Open a Subsystem

Open the Controller subsystem of the f14 model.

load_system('f14')
open_system('f14/Controller')

Open a Subsystem in New Tab in Existing Window

Open the f14 model and open the Controller subsystem in a new tab.

f14
open_system('f14/Controller','tab')

Open a Subsystem in a Separate Window

Open a subsystem in its own Simulink Editor window.

open_system('f14')
open_system('f14/Controller','window')

Open a Referenced Model

Open the model sldemo_mdlref_counter, which is referenced by the CounterA model block in sldemo_mdlref_basic.

open_system('sldemo_mdlref_basic')
open_system('sldemo_mdlref_basic/CounterA')

The referenced model opens in its own Simulink Editor window.
**Open Block Dialog Box**

Open the block parameters dialog box for the first Gain block in the Controller subsystem.

```matlab
load_system('f14')
open_system('f14/Controller/Gain')
```

**Run Block Open Callback Function**

Define an OpenFcn callback for a block and execute the block callback.

```matlab
f14
set_param('f14/Pilot','OpenFcn','disp(''Hello World!'')')
open_system('f14/Pilot','OpenFcn')
```

The words Hello World appear on the MATLAB Command Prompt.

**Open Masked Subsystem**

Open the contents of the masked subsystem Vehicle in the model `sf_car`.

```matlab
open_system('sf_car')
open_system('sf_car/Vehicle', 'force')
```

**Open Multiple Systems with One Command**

Create a cell array of two model names, `f14` and `vdp`. Open both models using `open_system` with the cell array name.

```matlab
models = {'f14','vdp'}
open_system(models)
```

**Input Arguments**

`obj` — Model, referenced model, library, subsystem, or block path character vector
Model, referenced model, library, subsystem, or block path, specified as a character vector. If the model is not on the MATLAB path, specify the full path to the model file. Specify the block or subsystem using its full name, e.g., f14/Controller/Gain, on an opened or loaded model. On UNIX systems, the fully qualified path name of a model can start with a tilde (~), signifying your home directory.

Data Types: char

sys — Model or library path
character vector

The full name or path of a model or library, specified as a character vector.

Data Types: char

sbsys — Subsystem path
character vector

The full name or path of a subsystem in an open or loaded model, specified as a character vector.

Data Types: char

blk — Block or subsystem path
character vector

The full name or path of a block or subsystem in an open or loaded model, specified as a character vector.

Data Types: char

See Also

close_system | load_system | new_system | save_system

Introduced before R2006a
**openDialog**

Open configuration parameters dialog

**Syntax**

openDialog(configObj)

**Arguments**

*configObj*

A configuration set (`Simulink.ConfigSet`) or configuration reference (`Simulink.ConfigSetRef`)

**Description**

`openDialog` opens a configuration parameters dialog box. If `configObj` is a configuration set, the dialog box displays the configuration set. If `configObj` is a configuration reference, the dialog box displays the referenced configuration set, or generates an error if the reference does not specify a valid configuration set. If the dialog box is already open, its window becomes selected.

**Examples**

The following example opens a configuration parameters dialog box that shows the current parameters for the current model. The parameter values derive from the active configuration set or configuration reference (configuration object). The code is the same in either case; the only difference is which type of configuration object is currently active.

```matlab
myConfigObj = getActiveConfigSet(gcs);
openDialog(myConfigObj);
```
See Also
attachConfigSet | attachConfigSetCopy | closeDialog | detachConfigSet |
getActiveConfigSet | getConfigSet | getConfigSets | setActiveConfigSet

Topics
“Manage a Configuration Set”
“Manage a Configuration Reference”

Introduced in R2006b
parsim

Simulate dynamic system multiple times in parallel or serial

Syntax

simOut = parsim(in)
simOut = parsim(in,'ShowSimulationManager','on')
simOut = parsim(in,Name,Value)

Description

simOut = parsim(in) simulates a model using the inputs specified in the SimulationInput object, in. The parsim command uses an array of SimulationInput objects to run multiple simulations.

simOut = parsim(in,'ShowSimulationManager','on') simulates a model in parallel using the inputs specified in the SimulationInput object and opens the Simulation Manager UI. For more information, see Simulation Manager.

simOut = parsim(in,Name,Value) simulates a model in parallel using the inputs specified in the SimulationInput object and the options specified as the Name,Value pair.

The parsim command uses the Parallel Computing Toolbox license to run the simulations in parallel. parsim runs the simulations in serial if a parallel pool cannot be created or if Parallel Computing Toolbox is not used.

Examples

Using parsim with Rapid Accelerator

Simulate the model, vdp, in rapid accelerator mode.

Load the model.
model = 'vdp';
load_system(model)

This step builds the Rapid Accelerator target

Simulink.BlockDiagram.buildRapidAcceleratorTarget(model);

Create a SimulationInput object and use setModelParameter method to set RapidAcceleratorUpToDateCheck to 'off'.

in = Simulink.SimulationInput(model);
in = in.setModelParameter('SimulationMode', 'rapid-accelerator');
in = in.setModelParameter('RapidAcceleratorUpToDateCheck', 'off');

Simulate the model.

out = parsim(in)

**Simulate Model in Parallel with parsim**

Simulate the model, CSTR, in parallel by sweeping over a variable. An array of SimulationInput objects is used to perform the sweep.

Specify sweep values.

FeedTempSweep = 250:10:300;

Create an array of SimulationInput objects.

```matlab
for i = length(FeedTempSweep):-1:1;
in(i) = Simulink.SimulationInput('CSTR');
in(i) = in(i).setVariable('FeedTemp0',FeedTempSweep(i));
end
```

Simulate the model in parallel.

out = parsim(in, 'ShowProgress', 'on')

[08-Jan-2018 14:10:43] Checking for availability of parallel pool...
Starting parallel pool (parpool) using the 'local' profile ...
connected to 6 workers.
[08-Jan-2018 14:11:12] Loading Simulink on parallel workers...
[08-Jan-2018 14:11:40] Configuring simulation cache folder on parallel workers...
[08-Jan-2018 14:11:40] Loading model on parallel workers...
[08-Jan-2018 14:11:48] Running simulations...
[08-Jan-2018 14:12:04] Completed 1 of 6 simulation runs
[08-Jan-2018 14:12:04] Completed 2 of 6 simulation runs
[08-Jan-2018 14:12:04] Completed 3 of 6 simulation runs
[08-Jan-2018 14:12:08] Completed 4 of 6 simulation runs
[08-Jan-2018 14:12:09] Completed 5 of 6 simulation runs
[08-Jan-2018 14:12:09] Completed 6 of 6 simulation runs
[08-Jan-2018 14:12:09] Cleaning up parallel workers...

out =

1x6 Simulink.SimulationOutput array

**Input Arguments**

in — Simulink.SimulationInput object used to simulate the model
object, array

A Simulink.SimulationInput object or an array of Simulink.SimulationInput objects that is used to specify changes to the model for a simulation.

Example: in = Simulink.SimulationInput('vdp')

**Name-Value Pair Arguments**

**Note** All parameters passed to parsim command are unrelated to the parameters that are used with the sim command. To pass to the parsim command, use the list of following input arguments

Specify optional comma-separated pairs of Name, Value arguments. Name is the argument name and Value is the corresponding value. Name and Value must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as Name1, Value1, ..., NameN, ValueN.

Example: 'ShowProgress', 'on'

2-449
**AttachedFiles — Files to attach to parallel pool**
cell array
Specified as a cell array of additional files to attach to the parallel pool.

**ShowProgress — Show the progress of the simulations**
'on'(default) | 'off'
Set to 'on', to see the progress of the simulations in the command window. The progress is hidden when set to 'off'.

**Note** When the progress is shown, a message 'Cleaning up parallel workers..' may be displayed before the completion of the last few simulations. This message does not depend on the completion of the simulations. Simulations complete when the outputs are fetched from the future. For more information, see `Simulink.Simulation.Future`.

**RunInBackground — Run simulations in background**
'off' (default) | 'on'
Set to 'on' to run simulations asynchronously, keeping the MATLAB command prompt available for use.

**SetupFcn — Function handle to run once per worker**
function handle
Specify a function handle to 'SetupFcn' to run once per worker before the start of the simulations.

Example: 'SetupFcn',@()simulinkproject('Modelex/Model_example.prj')

**Note** When `buildRapidAcceleratorTarget` is used in the SetupFcn and the model has external inputs specified, either set 'LoadExternalInput' to 'off' or ensure that the specified external input is available on the workers to prevent compilation error.

**CleanupFcn — Function handle to run once per worker after running simulations**
function handle
Specify a function handle to 'CleanupFcn' to run once per worker after the simulations are completed.
**ManageDependencies — Manage model dependencies**

'off' (default) | 'on'

When `ManageDependencies` is set to **on**, model dependencies are automatically sent to the parallel workers if required. If `ManageDependencies` is set to **off**, explicitly attach model dependencies to the parallel pool.

**UseFastRestart — Use fast restart**

'off' (default) | 'on'

When `UseFastRestart` is set to **on**, simulations run on the workers using fast restart.

When performing a parameter sweep varying properties of a Simscape component (e.g. mass and inertia of a Solid block in Simscape Multibody), specify the Simscape block parameter as run-time configurable. For more information, see “About Simscape Run-Time Parameters” (Simscape).

**Note** When using `parsim`, use the `UseFastRestart` option and not the `FastRestart` option. See “Get Started with Fast Restart” for more information.

**TransferBaseWorkspaceVariables — Transfer variables to the parallel workers**

'off' (default) | 'on'

When `TransferBaseWorkspaceVariables` is set to **on**, variables used in the model and everything defined in the base workspace are transferred to the parallel workers.

**ShowSimulationManager — Starts the Simulation Manager app**

'off' (default) | 'on'

When `ShowSimulationManager` is set to **on**, you can use the Simulation Manager App to monitor simulations.

**StopOnError — Stop simulations on errors**

'off' (default) | 'on'

Setting `StopOnError` to **on** stops the execution of simulations if an error is encountered.
Output Arguments

`simOut` — Simulation object containing logged simulation results

Array of `Simulink.SimulationOutput` objects that contains all of the logged simulation results. The size of the array is equal to the size of the array of `Simulink.SimulationInput` objects.

All simulation outputs (logged time, states, and signals) are returned in a single `Simulink.SimulationOutput` object. You define the model time, states, and output that is logged using the **Data Import/Export** pane of the Model Configuration Parameters dialog box. You can log signals using blocks such as the To Workspace and Scope blocks. The **Signal & Scope Manager** can directly log signals.

Extended Capabilities

**Automatic Parallel Support**
Accelerate code by automatically running computation in parallel using Parallel Computing Toolbox™.

If you have Parallel Computing Toolbox installed, then when you use `parsim`, MATLAB automatically opens a parallel pool of workers on your local machine. MATLAB runs the simulations across the available workers. Control parallel behavior with the parallel preferences, including scaling up to a cluster.

For details, see “Run Multiple Simulations”.

See Also

`ExternalInput` | `Simulation Manager` | `Simulink.Simulation.Future` | `Simulink.SimulationInput` | `applyToModel` | `cancel` | `fetchNext` | `fetchOutputs` | `setBlockParameter` | `setInitialState` | `setModelParameter` | `setPostSimFcn` | `setPreSimFcn` | `setVariable` | `validate` | `wait`
Topics

“Rapid Accelerator Simulations Using Parsim” on page 15-814
“Parallel Simulations Using Parsim: Test-Case Sweep” on page 15-799
“Parallel Simulations Using Parsim: Parameter Sweep in Normal Mode” on page 15-804
“Parallel Simulations Using Parsim: Parameter Sweep in Rapid Accelerator Mode” on page 15-809
“Run Multiple Simulations”
“Run Parallel Simulations Using parsim”
“Run Parallel Simulations”

Introduced in R2017a
**performanceadvisor**

Open Performance Advisor

**Syntax**

`performanceadvisor(model)`

**Description**

`performanceadvisor(model)` opens the Performance Advisor on the model or subsystem specified by `model`. If the specified model or subsystem is not open, this command opens it.

**Input Arguments**

`model`

A character vector specifying the name or handle to the model or subsystem.

**Examples**

**Open Performance Advisor**

Open Performance Advisor on the vdp example model.

`performanceadvisor('vdp')`
Performance Advisor opens the vdp model and opens Performance Advisor on the model.

**Alternatives**

In the Simulink Editor, in the **Debug** tab, click **Performance Advisor**.

**See Also**

**Topics**

“Improve Simulation Performance Using Performance Advisor”
“Perform a Quick Scan Diagnosis”
“Improve vdp Model Performance”

**Introduced in R2013a**
reload

(Not recommended) Reload project

**Note** `simulinkproject` is not recommended. Use `currentProject` or `openProject` instead. For more information, see “Compatibility Considerations”.

**Syntax**

`reload(proj)`

**Description**

`reload(proj)` reloads the project. Use `reload` when you want to run the project startup shortcuts.

**Examples**

**Reload Project**

Open the airframe project and create a project object.

```matlab
sldemo_slproject_airframe;
proj = simulinkproject;
```

When you want to run the startup shortcuts again, reload the project.

```matlab
reload(proj)
```

**Input Arguments**

`proj` — Project

project object
Project, specified as a project object already created with `simulinkproject` to manipulate a project at the command line.

**Compatibility Considerations**

**Simulink project API is not recommended**

*Not recommended starting in R2019a*

Starting in R2019a, instead of `simulinkproject` and related functions, use the `currentProject` or `openProject` functions in MATLAB. The Simulink project API will continue to be supported, but, after R2019a, new features might be available only if you use the new MATLAB project API. There are no plans to remove the Simulink project API at this time.

The new MATLAB project API is part of the R2019a functionality enabling you to use projects in MATLAB, with or without Simulink. You can now share projects with users who do not have Simulink.

**See Also**

**Functions**

`isLoaded | simulinkproject`

**Introduced in R2013a**
removeCategory

(Not recommended) Remove project category of labels

**Note** simulinkproject is not recommended. Use currentProject or openProject instead. For more information, see “Compatibility Considerations”.

**Syntax**

removeCategory(proj,categoryName)

**Description**

removeCategory(proj,categoryName) removes a category of labels, categoryName, from the project specified by proj.

**Examples**

**Remove Category**

Open the airframe project and create a project object.

```matlab
sldemo_slproject_airframe;
proj = simulinkproject;
```

Create a new category of labels.

```matlab
createCategory(proj,'Engineers','char');
```

Remove the new category of labels.

```matlab
removeCategory(proj,'Engineers');
```
A message appears warning you that you cannot undo the operation. Click **Continue**. You can configure warnings in the Preferences in the Project Tool.

**Input Arguments**

`proj` — Project

`project object`

Project, specified as a project object. Use `simulinkproject` to create a project object to manipulate a project at the command line.

`categoryName` — Name of category

`character vector`

Name of the category to remove, which exists in the project, specified as a character vector.

**Compatibility Considerations**

**Simulink project API is not recommended**

*Not recommended starting in R2019a*

Starting in R2019a, instead of `simulinkproject` and related functions, use the `currentProject` or `openProject` functions in MATLAB. The Simulink project API will continue to be supported, but, after R2019a, new features might be available only if you use the new MATLAB project API. There are no plans to remove the Simulink project API at this time.

The new MATLAB project API is part of the R2019a functionality enabling you to use projects in MATLAB, with or without Simulink. You can now share projects with users who do not have Simulink.

**See Also**

**Functions**

`createCategory` | `findCategory` | `simulinkproject`
Introduced in R2013a
removeFile

(Not recommended) Remove file from project

**Note** simulinkproject is not recommended. Use currentProject or openProject instead. For more information, see “Compatibility Considerations”.

**Syntax**

removeFile(proj,file)

**Description**

removeFile(proj,file) removes a file from the project proj.

**Examples**

**Remove File from Project**

Open the airframe project and create a project object.

```matlab
sldemo_slproject_airframe;
proj = simulinkproject;
```

Remove a file.

```matlab
removeFile(proj,'models/AnalogControl.slx')
```

Add the file back to the project.
addFile(proj,'models/AnalogControl.slx')

### Input Arguments

**proj** — Project

project object

Project, specified as a project object. Use `simulinkproject` to create a project object to manipulate a project at the command line.

**file** — Path of file

character vector | file object

Path of the file to remove relative to the project root folder, including the file extension, specified as a character vector or a file object returned by `findFile`. The file must be in the project.

Example: 'models/myModelName.slx'

### Compatibility Considerations

**Simulink project API is not recommended**

_Not recommended starting in R2019a_

Starting in R2019a, instead of `simulinkproject` and related functions, use the `currentProject` or `openProject` functions in MATLAB. The Simulink project API will continue to be supported, but, after R2019a, new features might be available only if you use the new MATLAB project API. There are no plans to remove the Simulink project API at this time.

The new MATLAB project API is part of the R2019a functionality enabling you to use projects in MATLAB, with or without Simulink. You can now share projects with users who do not have Simulink.
See Also

**Functions**
addFile | findFile | simulinkproject

*Introduced in R2013a*
removeLabel

(Not recommended) Remove label from project

**Note** simulinkproject is not recommended. Use currentProject or openProject instead. For more information, see “Compatibility Considerations”.

**Syntax**

removeLabel(category,labelName)
removeLabel(file,categoryName,labelName)
removeLabel(file,labelDefinition)

**Description**

removeLabel(category,labelName) removes the label from the specified category of labels in the currently loaded project.

removeLabel(file,categoryName,labelName) removes the specified label in the category categoryName from the file. Use this syntax to specify category and label by name.

removeLabel(file,labelDefinition) removes the specified label labelDefinition from the file. Before you can remove the label, you need to get the label from the file. Label property or by using findLabel.

**Examples**

**Remove a Label**

Open the airframe project and create a project object.

sldemo_slproject_airframe;
proj = simulinkproject;
Examine the first existing category.

```matlab
cat = proj.Categories(1)
cat = 
    Category with properties:
        Name: 'Classification'
        DataType: 'none'
        LabelDefinitions: [1x8 slproject.LabelDefinition]
```

Define a new label in the category.

```matlab
createLabel(cat,'Future');
```

Remove the new label.

```matlab
removeLabel(cat,'Future');
```

## Input Arguments

- **category** — Category of labels
category object

  Category of labels, specified as a category object. Get a category object from the `proj.Categories` property or by using `findCategory`.

- **labelName** — Name of label
character vector

  Name of the label to remove, specified as a character vector.

- **file** — File to detach label from
  file object

  File to detach the label from, specified as a file object. You can get the file object by examining the project’s Files property (`proj.Files`), or use `findFile` to find a file by name. The file must be within the root folder.

- **categoryName** — Name of category that contains label
  character vector
Name of the category that contains the label to remove, specified as a character vector.

**labelDefinition — Label to detach**

Label definition object

Name of the label to detach, specified as a label definition object returned by the file.Label property or findLabel.

**Compatibility Considerations**

**Simulink project API is not recommended**

*Not recommended starting in R2019a*

Starting in R2019a, instead of simulinkproject and related functions, use the currentProject or openProject functions in MATLAB. The Simulink project API will continue to be supported, but, after R2019a, new features might be available only if you use the new MATLAB project API. There are no plans to remove the Simulink project API at this time.

The new MATLAB project API is part of the R2019a functionality enabling you to use projects in MATLAB, with or without Simulink. You can now share projects with users who do not have Simulink.

**See Also**

**Functions**

addLabel | createLabel | findCategory | findLabel | simulinkproject

**Introduced in R2013a**
**replace_block**

Replace blocks in Simulink model

**Syntax**

replBlks = replace_block(sys, current, new)
replBlks = replace_block(sys, Name, Value, new)
replBlks = replace_block(____,'noprompt')

**Description**

replBlks = replace_block(sys, current, new) replaces the blocks current in the model sys with blocks of type new. You can use a block from a Simulink library or from another model as the replacement block.

Load the model sys before using this function. The function prompts you to select the blocks you want to replace from a list of blocks that match the current argument.

**Tip** Save the model before using this command.

replBlks = replace_block(sys, Name, Value, new) replaces the blocks that match the block parameters specified by the Name, Value pair arguments. You can also use find_system Name, Value pairs to qualify the search for blocks to replace.

Specify optional comma-separated pairs of Name, Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside quotes. You can specify several name and value pair arguments in any order as Name1, Value1, ..., NameN, ValueN.

With the replace_block function, you can use block parameter and value pairs. For a list of all the block parameters, see “Common Block Properties” on page 6-112 and “Block-Specific Parameters” on page 6-132.

To specify additional information about the search for blocks to replace, you use find_system Name, Value pairs before the block parameters. For example, you can use
CaseSensitive,'off' to make the search for blocks case insensitive or 'FollowLinks','on' to follow links into library links. See find_system for that list of Name,Value pairs.

replBlks = replace_block(___,'noprompt') replaces the blocks without prompting you to select them from a dialog box.

Examples

Replace Blocks in a Model

Replace blocks in the 'vdp' model.

Load the model 'vdp'.

load_system('vdp');

Replace Gain blocks with Integrator blocks.

RepNames = replace_block('vdp','Gain','Integrator');

A dialog box prompts you to select the blocks you want to replace.

With vdp/Mu selected in the dialog box, click OK.

Replace Scope blocks with To Workspace blocks.

RepNames = replace_block('vdp','Scope','simulink/Sinks/To Workspace');

A dialog box prompts you to select the blocks you want to replace.

With vdp/Scope selected in the dialog box, click OK.

Replace Blocks in a Subsystem Using Parameter Values

Replace blocks in the Unlocked subsystem of the sldemo_clutch model. Replace blocks whose Gain parameter is set to bv.

Load the model sldemo_clutch.
load_system('sldemo_clutch');

In the 'Unlocked' subsystem, replace blocks whose Gain value is bv with Integrator blocks.

replace_block('sldemo_clutch/Unlocked','Gain','bv','Integrator');

A dialog box prompts you to select the blocks to replace.

With sldemo_clutch/Unlocked/VehicleDamping selected in the dialog box, click OK.

**Replace Blocks Without Dialog Box**

Load the model f14.

load_system('f14')

Replace Gain blocks with Integrator blocks. The command returns the blocks it found to replace and replaces the blocks.

repl = replace_block('f14','Gain','Integrator','noprompt')

repl = 13x1 cell array
   {'f14/Aircraft...'}    }
   {'f14/Aircraft...'}    }
   {'f14/Aircraft...'}    }
   {'f14/Aircraft...'}    }
   {'f14/Controller/Gain'}
   {'f14/Controller/Gain2'}
   {'f14/Controller/Gain3'}
   {'f14/Gain'}
   {'f14/Gain1'}
   {'f14/Gain2'}
   {'f14/Gain5'}
   {'f14/Nz pilot...'}    }
   {'f14/Nz pilot...'}    }

2-469
Use find_system Pairs with replace_block

Select a block that is a library link. Follow the library links and replace Gain blocks with Integrator blocks within them.

```
replace_block(gca, 'FollowLinks', 'on', 'BlockType', 'Gain', 'Integrator', 'noprompt')
```

Input Arguments

**sys — Model or subsystem whose blocks to replace**

character vector

Name of model whose blocks to replace, specified as a character vector. If you specify a model, the command replaces all blocks that match in the model. If you specify a subsystem, the command replaces blocks in that subsystem and below.

Example: 'vdp', 'sldemo_fuelsys/fuel_rate_control'

**current — Type of block to replace**

BlockType value | MaskType value

Type of block to replace, specified as a BlockType or MaskType value. To find out the block type, select the block and, at the command prompt, enter:

```
get_param(gca,'BlockType')
```

For masked blocks, to find out the mask type, select the block and enter:

```
get_param(gca,'MaskType')
```

**new — Block to replace current blocks**

BlockType value | MaskType value | library path | block path name from a model

Block to replace the current block, specified in one of these forms:

- BlockType value of the replacement block. Specifying this value uses a library block as the replacement block.
- MaskType value of the replacement block. Specifying this value uses a library block as the replacement block.
- Library path of the replacement block, for example, 'simulink/Sinks/To Workspace'. Hover over the block in the library to see the library path.
• Block path name of a block from a different model, for example, 'vdp/Mu'. Use this value to reuse an instance of a block from another model in your model.

Output Arguments

replBlks — Blocks returned by the current argument

cell array of character vectors

Blocks returned by the current argument, returned as a cell array of character vectors. The function returns the values regardless of whether you complete the replacement.

See Also

find_system | get_param

Topics

“Common Block Properties” on page 6-112
“Block-Specific Parameters” on page 6-132

Introduced before R2006a
save_system

Save Simulink model

Syntax

filename = save_system
filename = save_system(sys)
filename = save_system(sys,newsys)
filename = save_system(sys,newsys,Name,Value)

Description

filename = save_system saves the current top-level model. If the model was not previously saved, save_system creates a file in the current folder.

To save a subsystem, instead use Simulink.SubSystem.copyContentsToBlockDiagram to copy the subsystem contents to a new model. You can then save that model using save_system. See Simulink.SubSystem.copyContentsToBlockDiagram.

If you set the model UpdateHistory property to UpdateHistoryWhenSave, no dialog box prompt appears when you use save_system to save the model. If you want to update the comment, use the 'ModifiedComment' parameter with set_param before saving, for example:

set_param('mymodel','ModifiedComment','Here is my comment.')

filename = save_system(sys) saves the model sys. The model must be open or loaded.

filename = save_system(sys,newsys) saves the model to a new file newsys. If you do not specify an extension, then save_system uses the file format specified in your Simulink preferences.

filename = save_system(sys,newsys,Name,Value) saves the system with additional options specified by one or more Name,Value pair arguments. To use Name,Value pairs without saving to a new file, use [] for newsys.
Examples

Save Named Model

Create a model.

\begin{verbatim}
new_system('newmodel')
\end{verbatim}

Save the model.

\begin{verbatim}
save_system('newmodel')
\end{verbatim}

Save Model with Another Name

Open the model vdp. Save it to a model named myvdp in the current folder. Without a file
extension, the function saves the model using the format specified in your Simulink
preferences.

\begin{verbatim}
open_system('vdp')
save_system('vdp','myvdp')
\end{verbatim}

After you save the model by another name, the model is no longer open under its original
name. Open vdp again and save it as an .mdl file in the current folder.

\begin{verbatim}
open_system('vdp')
save_system('vdp','mynewvdp.mdl')
\end{verbatim}

Return Error If Name Exists

Save a model with a new name and return an error if something with this name exists on
the MATLAB path. In this case, save_system displays an error because max is the name
of a MATLAB function. The model is not saved.

\begin{verbatim}
open_system('vdp')
save_system('vdp','max','ErrorIfShadowed',true)
\end{verbatim}

Error using save_system (line 38)
The model 'vdp' cannot be saved with the new name 'max', because this name is
shading another name on the MATLAB path or in the workspace. Choose another name, or do not use the option 'ErrorIfShadowed'

Save Model with Options

Suppose that you have a model named mymodel. Open the model and save it to a model named newmodel. Also save the model workspace, break links to user-defined library blocks, and overwrite if the file has changed on disk,

```matlab
open_system('mymodel')
save_system('mymodel','newmodel','SaveModelWorkspace',true,'BreakUserLinks',true,'OverwriteIfChangedOnDisk',true)
```

Save Model to Same Name and Use Options

Save the model mymodel, breaking links to user-defined library blocks in the model.

```matlab
save_system('mymodel',[],'BreakUserLinks',true)
```

Input Arguments

**sys — Name of model to save**
character vector | cell array of character vectors | string array | handle | array of handles

Name of model to save, specified as a character, cell array of character vectors, string array, handle, or array of handles. Do not use a file extension.

**newsys — File to save to**
character vector | cell array of character vectors | string array | []

File to save to, specified as a character vector, cell array of character vectors, string array, or, to use Name, Value pairs without changing the file name, []. You can specify a model name in the current folder or the full path name, with or without an extension.

With no an extension, `save_system` saves to the file format specified in your Simulink preferences. Possible model extensions are .slx and .mdl. With the 'ExportToXML' option, use the extension .xml.

For information on rules for naming models, see “Model Names”.

2-474
**Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name, Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example:
```
save_system('mymodel', 'newmodel', 'SaveModelWorkspace', true, 'BreakUserLinks', true, 'OverwriteIfChangedOnDisk', true)
```

**AllowPrompt — Allow dialog box prompts**

`false` (default) | `true` | `'on'` | `'off'`

Option to allow dialog box prompts, specified as `true`, `false`, `'on'`, or `'off'`. By default, warnings and error messages appear at the command line.

**BreakAllLinks — Replace links to library blocks**

`false` (default) | `true` | `'on'` | `'off'`

Option to, in the saved file, replace links to library blocks with copies of the library blocks, specified as `true`, `false`, `'on'`, or `'off'`. This option affects user-defined blocks and Simulink library blocks.

**Caution** The `'BreakAllLinks'` option can result in compatibility issues when upgrading to newer versions of Simulink. For example:

- Any masks on top of library links to Simulink S-functions do not upgrade to the new version of the S-function.
- Any library links to masked subsystems in a Simulink library do not upgrade to the new subsystem behavior.
- Any broken links prevent the library forwarding mechanism from upgrading the link.

If you saved a model with broken links to built-in libraries, use the Upgrade Advisor to scan the model for out-of-date blocks. Then upgrade the Simulink blocks to their current versions.

**BreakUserLinks — Replace links to user-defined blocks**

`false` (default) | `true` | `'on'` | `'off'`
Option to, in the saved file, replace links to user-defined library blocks with copies of the library blocks, specified as true, false, 'on', or 'off'.

**BreakToolboxLinks — Replace links to built-in library block**
false (default) | true | 'on' | 'off'

Option to, in the saved file, replace links to built-in library blocks with copies of the library blocks, specified as true, false, 'on', or 'off'. This option affects Simulink library blocks and blocks from libraries supplied with MathWorks toolboxes or blocksets.

**ErrorIfShadowed — Return an error if name exists**
false (default) | true | 'on' | 'off'

Option to return an error if the new name exists on the MATLAB path or workspace, specified as true, false, 'on', or 'off'.

**ExportToXML — Export model to XML format**
false (default) | true | 'on' | 'off'

Option to export the model to a file in a simple XML format, specified as true, false, 'on', or 'off'. Specify the full name of the file, including the .xml extension. The block diagram in memory does not change and no callbacks execute. Use this option without any other Name,Value pair arguments. This option warns and will be removed in a future release.

Example: `save_system('mymodel','exportfile.xml','ExportToXML',true)`

**ExportToVersion — MATLAB release name to export to**
character vector | string scalar

MATLAB release name to export to, specified in either of these forms (not case sensitive). You can export to seven years of previous releases.

- Release name, for example, 'R2013B', 'R2016B'
- Release name, followed by an underscore and then the extension, for example, 'R2016A_SLX', 'R2014A_MDL'. If you do not specify an extension, you export to the file format specified in your Simulink preferences.

`save_system` exports the system such that the specified Simulink version can load it. If the system contains functionality not supported by the specified Simulink version, the command removes the functionality in the exported file. It also replaces unsupported blocks with empty masked subsystem blocks colored yellow. As a result, the exported system might generate different results.
Alternatively, use `Simulink.exportToVersion` or, interactively, the Export to Previous Version dialog box.

**OverwriteIfChangedOnDisk — Overwrite file**

false (default) | true | 'on' | 'off'

Option to overwrite the file on disk even if it has been modified since the system was loaded, specified as true, false, 'on', or 'off'. By default, if the file changed on disk since the model was loaded, `save_system` displays an error to prevent the changes on disk from being overwritten.

You can control whether `save_system` displays an error if the file has changed on disk using a Simulink preference. In the **Model File** pane of the Simulink Preferences dialog box, under **Change Notification**, select **Saving the model**. This preference is on by default.

**SaveDirtyReferencedModels — Save referenced models with unsaved changes**

false (default) | true | 'on' | 'off'

Option to save referenced models that contain unsaved changes while also saving the model, specified as true, false, 'on', or 'off'. By default, attempting to save a model that contains unsaved referenced models return an error.

**SaveModelWorkspace — Save model workspace**

false (default) | true | 'on' | 'off'

Option to save the contents of the model workspace, specified as true, false, 'on', or 'off'. The model workspace **DataSource** must be a MAT-file. If the data source is not a MAT-file, `save_system` does not save the workspace. See “Specify Source for Data in Model Workspace”.

**Output Arguments**

**filename — Name of saved file**

character vector | cell array of character vectors

Full name of saved file, returned as a character vector or a cell array of character vectors.
See Also
Simulink.exportToVersion | close_system | new_system | open_system

Topics
“Save a Model”

Introduced before R2006a
**Schedule Editor**

View and edit the schedule of model components (partitions)

**Description**

The Schedule Editor is a scheduling tool that represents the components in the model known as partitions, the data connections between them, and the execution order of those partitions.

Partitions are the components of the model that execute independently as tasks. The data connections between the partitions show the flow of the data between those partitions. The scheduling of these partitions is based on the rates and the events in the model. This schedule is shown in the **Execution Order** table in the Schedule Editor.

Using the Schedule Editor, you can:

- Create partitions and specify their execution order.
- Edit and analyze the schedule of the executable partitions without disturbing the structure of the model.
- Visualize how Simulink executes partitions.

Changes made in the Schedule Editor affect both, simulation and code generation.

**Using the Schedule Editor**

The Schedule Editor consists of two parts representing two different views of partitions in the model:

- A graph that shows the partitions and the data connections between them.
• A table that shows the order in which the partitions execute.

<table>
<thead>
<tr>
<th>Order</th>
<th>Name</th>
<th>Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>D1</td>
<td>0.1</td>
</tr>
<tr>
<td>2</td>
<td>Component.D1</td>
<td>0.1</td>
</tr>
<tr>
<td>3</td>
<td>Component.P1</td>
<td>0.1</td>
</tr>
<tr>
<td>4</td>
<td>Component.P2</td>
<td>0.1</td>
</tr>
<tr>
<td>5</td>
<td>Component.D2</td>
<td>0.2</td>
</tr>
</tbody>
</table>

Changing one of the views impacts the other.

To use the Schedule Editor, select **View > Schedule Editor**. If the model is already partitioned, you can open the Schedule Editor by clicking the badge, which appears above the blocks. To see the default partitions present in the model in the Schedule Editor, update the diagram by clicking the icon on the toolstrip or by selecting **Ctrl + D**. As you create partitions in the model and update the diagram, partitions appear in the Schedule Editor.

To check how the partitions map to the model, right-click the partitions and select **Show Source**. The Simulink model window appears with every block corresponding to the partition highlighted.
Execution Order

The Execution Order shows the order in which the partitions execute. To change the execution order, you can drag and drop the partitions. You can also use the Up and Down arrows on the toolstrip. Partitions are sorted based on their rates. You can only reorder the partitions with the same rate. Clicking a partition in the Execution Order, highlights the corresponding partition in the graph. On changing the execution order, the connections that are affected by this specified change get highlighted.

<table>
<thead>
<tr>
<th>Order</th>
<th>Name</th>
<th>Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>D1</td>
<td>0.1</td>
</tr>
<tr>
<td>3</td>
<td>Component.P2</td>
<td>0.1</td>
</tr>
<tr>
<td>2</td>
<td>Component.D1</td>
<td>0.1</td>
</tr>
<tr>
<td>4</td>
<td>Component.U2</td>
<td>0.2</td>
</tr>
<tr>
<td>5</td>
<td>Component.P1</td>
<td>0.2</td>
</tr>
</tbody>
</table>

Connections

Connections between the partitions show data dependencies. You can right-click the connections between the partitions to change the constraints on data connections. The different types of connections illustrate how the partitions behave with each other.

The types of connections are:

- **Dependency** — Indicates that the source always runs before the destination. The dependency connection is a solid line.

- **Delay** — Indicates that the destination runs before the source. When the destination runs before the source, a scheduling delay is introduced. The delay connection is a dashed line.

You can put these types of constraints on connections:
• **Allow Delay** — Inserts a delay when required. When you specify this constraint for a connection, Simulink inserts a delay for that connection only when necessary. The unlock icon on the connections signifies an allowed delay. When you select this constraint on a connection, Simulink prefers these connections to be turned into a delay if necessary over other connections.

This constraint is displayed as one of these options.

![Unlock Icon](image)

• **Prevent Delay** — Prevents delay from being inserted in the connection. When you specify this constraint for a connection, Simulink ensures that the connection is a dependency. The lock icon on the connection indicates that the connection is locked as a dependency and is not changed to a delay.

![Lock Icon](image)

**Open the Schedule Editor**

• Simulink: In the **View**, select **Schedule Editor**.
• Simulink model: Click the badge on the partitioned blocks.
• Keyboard shortcut: Select **Alt+V+S**.

**Examples**

• “Schedule an Export-Function Model Using the Schedule Editor”
• “Schedule a Rate-Based Model Using the Schedule editor”
Parameters

Manage Partitions

**Partition Name — Name of the partitions**
character vector

Name of the partitions created. Double-click the name to edit and use the enter key to save the partition name.

**Sample Time — Sample time for partitions to execute**
discrete value | variable

Sample time at which the partitions execute. The sample time is a discrete value or a defined variable that contains a discrete value.

Property Inspector

**Hit times — Times for aperiodic partitioned to execute**
vector

Times at which the aperiodic partitions execute, specified as a vector of discrete values.

**Connections — Data connections between partitions**
Auto (default) | Delay | Dependency | Prevent Delays | Allow Delays

Dependency between the partitions is shown as a solid line. A delay between the partitions is shown as a dashed line. The arrow tail and head denote the source and destination, respectively.

Specify constraints on the connections by right-clicking the lines and selecting the constraint type.

More About

Execution Order

By default, the **Execution Order** table is expanded. **Execution Order** shows the order of partition execution. Edit the order by dragging the partitions or using the **Up** and **Down** arrows on the toolstrip.
Manage Partitions

By default, the Manage Partitions panel is collapsed. Expand the panel to create partitions, edit names and/or sample times of the existing partitions, and remove existing partitions. Changes made in the Manage Partitions panel take effect after updating the diagram.

Property Inspector

The Property Inspector shows information about the selected partition. It shows the rate at which the partition executes and the connections on that partition. Enter the Hit times when you select an aperiodic partition in the Property Inspector.

Arrange

The Arrange button organizes the partition graph.

Layout

By default, the Execution Order panel and the Property Inspector panel are expanded. From the Layout drop-down, select the preferred panels to be expanded as per preference. Clearing the panels collapses them.

See Also

“Create Partitions” | “Generate Code from a Partitioned Model”

Topics

“Schedule an Export-Function Model Using the Schedule Editor”
“Schedule a Rate-Based Model Using the Schedule editor”

Introduced in R2019a
set_param

Set system and block parameter values

Syntax

set_param(Object,ParameterName,Value,...ParameterNameN,ValueN)

Description

set_param(Object,ParameterName,Value,...ParameterNameN,ValueN) sets the parameter to the specified value on the specified model or block object.

When you set multiple parameters on the same model or block, use a single set_param command with multiple pairs of ParameterName, Value arguments, rather than multiple set_param commands. This technique is efficient because using a single call requires evaluating parameters only once. If any parameter names or values are invalid, then the function doesn’t set any parameters.

Tips:

• If you make multiple calls to set_param for the same block, then specifying the block using a numeric handle is more efficient than using the full block path. Use getSimulinkBlockHandle to get a block handle.
• If you use matlab -nodisplay to start a session, you cannot use set_param to run your simulation. The -nodisplay mode does not support simulation using set_param. Use the sim command instead.
• After you set parameters in the MATLAB workspace, to see the changes in a model, update the diagram.

    set_param(model,'SimulationCommand','Update')

For parameter names, see:

• “Model Parameters” on page 6-2
• “Block-Specific Parameters” on page 6-132
• “Common Block Properties” on page 6-112

Examples

Set Model Configuration Parameters for a Model

Open vdp and set the Solver and StopTime parameters.

```matlab
vdp
set_param('vdp','Solver','ode15s','StopTime','3000')
```

Set Model Configuration Parameters for Current Model

Open a model and set the Solver and StopTime parameters. Use bdroot to get the current top-level model.

```matlab
vdp
set_param(bdroot,'Solver','ode15s','StopTime','3000')
```

Set a Gain Block Parameter Value

Open vdp and set a Gain parameter value in the Mu block.

```matlab
vdp
set_param('vdp/Mu','Gain','10')
```

Set Position of Block

Open vdp and set the position of the Fcn block.

```matlab
vdp
set_param('vdp/Fcn','Position',[50 100 110 120])
```
Set Position of Block Using a Handle

Set the position of the Fcn block in the vdp model.

Use `getSimulinkBlockHandle` to load the vdp model if necessary (by specifying true), and get a handle to the Fcn block. If you make multiple calls to `set_param` for the same block, then using the block handle is more efficient than specifying the full block path as a character vector.

```matlab
cfcnblockhandle = getSimulinkBlockHandle('vdp/Fcn',true);
```

You can use the block handle in subsequent calls to `get_param` or `set_param`. If you examine the handle, you can see that it contains a double. Do not try to use the number of a handle alone (e.g., 5.007) because you usually need to specify many more digits than MATLAB displays. Instead, assign the handle to a variable and use that variable name to specify a block.

Use the block handle with `set_param` to set the position.

```matlab
set_param(fcnblockhandle,'Position',[50 100 110 120])
```

**Input Arguments**

**Object — Name or handle of a model or block**

- character vector | string scalar | handle

Handle or name of a model or block, specified as a numeric handle or a character vector. A numeric handle must be a scalar. You can also set parameters of lines and ports, but you must use numeric handles to specify them.

**Tip** If you make multiple calls to `set_param` for the same block, then specifying a block using a numeric handle is more efficient than using the full block path with `set_param`. Use `getSimulinkBlockHandle` to get a block handle. Do not try to use the number of a handle alone (e.g., 5.007) because you usually need to specify many more digits than MATLAB displays. Assign the handle to a variable and use that variable name to specify a block.

Example: 'vdp/Fcn'
**ParameterName — Model or block parameter name**
character vector | string scalar

Model or block parameter name, specified as a character vector or a string scalar. Some parameter names are case sensitive.
Example: 'Solver'
Example: 'SimulationCommand'
Example: 'Position'

**Value — Value of the specified parameter**
any data type, depending on the parameter

Model or block parameter value, specified in the format determined by the parameter type. Some parameter values are case sensitive. Values are often character vectors, but they can also be numeric, arrays, and other types. Many block parameter values are specified as character vectors, but two exceptions are these parameters: Position, specified as a vector, and UserData, which can be any data type.

**See Also**
bdroot | gcb | gcs | getSimulinkBlockHandle | get_param | new_system | open_system

**Topics**
“Associating User Data with Blocks”
“Use MATLAB Commands to Change Workspace Data”
“Run Simulations Programmatically”
“Model Parameters” on page 6-2
“Block-Specific Parameters” on page 6-132
“Common Block Properties” on page 6-112

**Introduced before R2006a**
**setActiveConfigSet**

Specify model's active configuration set or configuration reference

**Syntax**

```
setActiveConfigSet(model, configObjName)
```

**Arguments**

- **model**
  - The name of an open model, or `gcs` to specify the current model
- **configObjName**
  - The name of a configuration set (`Simulink.ConfigSet`) or configuration reference (`Simulink.ConfigSetRef`)

**Description**

`setActiveConfigSet` specifies the active configuration set or configuration reference (configuration object) of `model` to be the configuration object specified by `configObjName`. If no such configuration object is attached to the model, an error occurs. The previously active configuration object becomes inactive.

**Examples**

The following example makes `DevConfig` the active configuration object of the current model. The code is the same whether `DevConfig` is a configuration set or configuration reference.

```
setActiveConfigSet(gcs, 'DevConfig');
```
See Also
attachConfigSet | attachConfigSetCopy | closeDialog | detachConfigSet |
getActiveConfigSet |getConfigSet | getConfigSets | openDialog

Topics
“Manage a Configuration Set”
“Manage a Configuration Reference”

Introduced before R2006a
sfix

Create Simulink.NumericType object describing signed fixed-point data type

Syntax

\[ a = \text{sfix(WordLength)} \]

Description

\text{sfix(WordLength)} \text{ returns a Simulink.NumericType object that describes a signed fixed-point number with the specified word length and unspecified scaling.}

\textbf{Note} \text{sfix is a legacy function. In new code, use \text{fixdt} instead. In existing code, replace sfix(WordLength) with fixdt(1,WordLength).}

Examples

Define a 16-bit signed fixed-point data type.

\[ a = \text{sfix(16)} \]

\[ a = \]

\text{NumericType with properties:}

\begin{itemize}
  \item \text{DataTypeMode: 'Fixed-point: unspecified scaling'}
  \item \text{Signedness: 'Signed'}
  \item \text{WordLength: 16}
  \item \text{IsAlias: 0}
  \item \text{DataScope: 'Auto'}
  \item \text{HeaderFile: ''}
  \item \text{Description: ''}
\end{itemize}
See Also
Simulink.NumericType | fixdt | float | sfrac | sint | ufix | ufrac | uint

Introduced before R2006a
sfrac

Create Simulink.NumericType object describing signed fractional data type

Syntax

a = sfrac(WordLength)
a = sfrac(WordLength, GuardBits)

Description

sfrac(WordLength) returns a Simulink.NumericType object that describes the data type of a signed fractional data type with a word size given by WordLength.

sfrac(WordLength, GuardBits) returns a Simulink.NumericType object that describes the data type of a signed fractional number. The total word size is given by WordLength with GuardBits bits located to the left of the binary point.

The most significant (leftmost) bit is the sign bit. The default binary point for this data type is assumed to lie immediately to the right of the sign bit. If guard bits are specified, they lie to the left of the binary point and to right of the sign bit. For example, the structure for an 8-bit signed fractional data type with 4 guard bits is:

Note sfrac is a legacy function. In new code, use fixdt instead. In existing code, replace sfrac(WordLength,GuardBits) with fixdt(1,WordLength, (WordLength-1-GuardBits)) and sfrac(WordLength) with fixdt(1,WordLength,(WordLength-1)).
Examples

Define an 8-bit signed fractional data type with 4 guard bits. Note that the range of this data type is $-2^4 = -16$ to $(1 - 2^{(1 - 8)})2^4 = 15.875$.

```matlab
a = sfrac(8,4)
```

```
NumericType with properties:

    DataTypeMode: 'Fixed-point: binary point scaling'
    Signedness: 'Signed'
    WordLength: 8
    FractionLength: 3
    IsAlias: 0
    DataScope: 'Auto'
    HeaderFile: ''
    Description: ''
```

See Also

Simulink.NumericType | fixdt | float | sfix | sint | ufix | ufrac | uint

Introduced before R2006a
**showblockdatatypetable**

Display HTML page of Simulink block data type support

**Syntax**

showblockdatatypetable

**Description**

showblockdatatypetable displays an HTML page of the Simulink block data type support. This table that lists the data types that Simulink blocks support.

**Examples**

**Display Support of Simulink Block Data Types**

Display an HTML page of the Simulink block data type support.

showblockdatatypetable

**Tips**

To open the Block Support Table using a block, double-click the Block Support Table block in the block library or add it to a model and then double-click it.

**See Also**

Block Support Table

**Topics**

“Data Types Supported by Simulink”
Introduced before R2006a
showunitslist

Show built-in units, physical quantities, and unit systems supported by Simulink

Syntax

showunitslist

Description

showunitslist opens a web browser that contains the list of built-in units, physical quantities, and unit systems supported by Simulink.

Examples

Show Built-in Units, Physical Quantities, and Unit Systems

Show built-in units, physical quantities, and unit systems supported by Simulink.

showunitslist

This function opens a web browser displaying the allowed units.

See Also

Topics

“Unit Specification in Simulink Models”

Introduced in R2016a
signalbuilder

(Not recommended) Create and access Signal Builder blocks

**Syntax**

```matlab
[time, data] = signalbuilder(block)
[time, data, signames] = signalbuilder(block)
[time, data, signames, groupnames] = signalbuilder(block)

block = signalbuilder(path,'create',time,data)
block = signalbuilder(path,'create',time,data,signames,groupnames)
block = signalbuilder(path,'create',time,data,signames,groupnames, vis)
block = signalbuilder(path,'create',time,data,signames,groupnames, vis,pos)
block = signalbuilder(path,'create',time,data,signames,groupnames, vis,pos,{openui openmodel})

block = signalbuilder(block,'appendgroup',time,data,signames, groupnames)
signalbuilder(block,'appendgroup',ds)
signalbuilder(block,'appendgroup', [ds1 ...dsN])
signalbuilder(block,'appendsignal',time,data,signames)
signalbuilder(block,'showsignal',signal,group)
signalbuilder(block,'hidesignal',signal, group)

[time, data] = signalbuilder(block,'get',signal,group)
ds = signalbuilder(block,'get',group)
[ds, ...dsN] = signalbuilder(block,'get',group)
signalbuilder(block,'set',signal,group,time,data)
signalbuilder(block,'set',group,ds)
signalbuilder(block,'set',group,[ds1 ...dsN])

index = signalbuilder(block,'activegroup')
```
[index, activeGroupLabel] = signalbuilder(block,'activegroup')
signalbuilder(block,'activegroup',index)
signalbuilder(block,'annotategroup')
signalbuilder(block,'print',config,printArgs)
figh = signalbuilder(block,'print',config,'figure')

**Description**

\[\text{[time, data]} = \text{signalbuilder(block)}\] returns the time and data of the Signal Builder block, block.

\[\text{[time, data, signames]} = \text{signalbuilder(block)}\] returns signal names, signames, and time and data.

\[\text{[time, data, signames, groupnames]} = \text{signalbuilder(block)}\] returns the signal names, signames, and group names, groupnames, and time and data.

\[\text{block = signalbuilder(path,'create',time, data)}\] creates a new Signal Builder block at path with specified time and data. Signal and group names are default.

\[\text{block = signalbuilder(path,'create',time, data, signames, groupnames)}\] creates a new Signal Builder block at path with specified time and data, signal names, and group names.

\[\text{block = signalbuilder(path,'create',time, data, signames, groupnames, vis)}\] creates a new Signal Builder block and sets the visible signals in each group based on the values of the matrix vis.

\[\text{block = signalbuilder(path,'create',time, data, signames, groupnames, vis,pos)}\] creates a new Signal Builder block and sets the block position to pos.

If you create signals that are smaller than the display range or do not start from 0, the Signal Builder block extrapolates the undefined signal data. It does so by holding the final value.

\[\text{block = signalbuilder(path,'create',time, data, signames, groupnames, vis,pos,\{openui openmodel\})}\] creates a new Signal Builder block and opens or invisibly loads the model and Signal Builder block window.
block = signalbuilder(block,'appendgroup',time,data,signames, groupnames) appends new groups (groupnames) containing signames to the Signal Builder block, block. The time and data arguments must have the same number of signals as the existing block.

signalbuilder(block,'appendgroup',ds) appends one dataset.

signalbuilder(block,'appendgroup',[ds1 ...dsN]) appends N datasets.

signalbuilder(block,'appendsignal',time,data,signames) appends new signals to all signal groups in the Signal Builder block, block. You can append either the same signals to all groups, or append different signals to different groups. Regardless of which signals you append, append the same number of signals to all the groups. Append signals to all the groups in the block; you cannot append signals to a subset of the groups. Correspondingly, provide time and data arguments for either one group (append the same information to all groups) or different time and data arguments for different groups.

signalbuilder(block,'showsignal',signal,group) makes signals that are hidden from the Signal Builder block visible. By default, signals in the current active group are visible when created.

signalbuilder(block,'hidesignal',signal, group) makes signals, signal, hidden from the Signal Builder block. By default, all signals are visible when created.

[time,data] = signalbuilder(block,'get',signal,group) gets the time and data values for the specified signal(s) and group(s).

ds = signalbuilder(block,'get',group) gets one or more datasets for one requested Signal Builder group. r gets N datasets for N requested Signal Builder groups.

[ds, ...dsN] = signalbuilder(block,'get',group) gets N datasets for N requested Signal Builder groups.

signalbuilder(block,'set',signal,group,time,data) sets the time and data values for the specified signal and group. Use empty values of time and data to remove groups and signals. To remove a signal group, you must also remove all the signals in that group in the same command.

signalbuilder(block,'set',group,ds) sets one dataset for the requested Signal Builder group. Specifying an empty dataset deletes the groups specified in group.
signalbuilder(block, 'set', group, [ds1 ... dsN]) sets N datasets for N requested groups.

index = signalbuilder(block, 'activegroup') gets the index of the active group.

[index, activeGroupLabel] = signalbuilder(block, 'activegroup') gets the label value of the active group.

signalbuilder(block, 'activegroup', index) sets the active group to indexed active group.

signalbuilder(block, 'annotategroup') controls the display of the current group name on the mask of the Signal Builder block. 'annotategroup' takes one of these values:

- 'on' — Shows the current group name
- 'off' — Hides the current group name

signalbuilder(block, 'print', config, printArgs) prints the currently active signal group or the signal group that config specifies. Use the config to customize the printed appearance of a signal group.

figh = signalbuilder(block, 'print', config, 'figure') prints the currently active signal group or the signal group that config specifies to a new hidden figure handle, figh.

**Examples**

**Create Signal Builder Block**

Create a Signal Builder block in a new model editor window.

block = signalbuilder([], 'create', [0 5], [[2 2]; [0 2]]);

Get Signal Builder data from this block.

[time, data, signames, groupnames] = signalbuilder(block)

time =

2x1 cell array
data =
{1×2 double}  
{1×2 double}

data =
2×1 cell array
{1×2 double}  
{1×2 double}

signames =
1×2 cell array
{'Signal 1'} {'Signal 2'}

groupnames =
1×1 cell array
{'Group 1'}

The Signal Builder block contains two signals in one group. Alter the second signal in the group.

signalbuilder(block, 'set', 2, 1, [0 5], [2 0]);

To make this same change using the signal name and group name:

signalbuilder(block, 'set', 'Signal 2', 'Group 1', [0 5], [2 0])

Create Signal Builder Block and Delete Group

Create a Signal Builder block with two signal groups and delete one of the groups.

block = signalbuilder([], 'create', [0 2], {{0 1},[1 0]});

The Signal Builder block has two groups, each of which contains a signal.

To delete the second group, also delete its signal.
Create Signal Builder Block with Two Groups

Create a Signal Builder block with two groups, each of which contains three signals.

```matlab
block = signalbuilder([], 'create', [0 1], ...
    {{[0 0],[1 1];[1 0],[0 1];[1 1],[0 0]}});
```

Create Signal Builder Block and Append Signal Groups

Create a Signal Builder block in a new model editor window.

```matlab
block = signalbuilder([], 'create', {
    [0 10],[0 20]
}, {
    [6 -6],...
    [2 5]});
```

The Signal Builder block has two groups. Each group contains one signal.

Append a new signal group to the existing block.

```matlab
block = signalbuilder(block, 'appendgroup', [0 30], [10 -10]);
```

Append a new signal, sig3, to all groups.

```matlab
signalbuilder(block, 'appendsignal', [0 30], [0 10], 'sig3');
```

Create Signal Builder Block and Delete Signal Group

Create a Signal Builder block in a new model editor window.

```matlab
time = [0 1];
data = {{[0 0],[1 1];[1 0],[0 1];[1 1],[0 0]};
block = signalbuilder([], 'create', time, data);
```

The Signal Builder block has two groups. Each group contains three signals.

Delete the second group. To delete a signal group, also delete all the signals in the group.
Create Signal Builder Block and Hide Signal

Create a Signal Builder block in a new model editor window and hide signal.

```matlab
block = signalbuilder([], 'create', [0, 5], {[2 2];[0 2]});
```

The Signal Builder block has one group that contains two signals.

Hide the signal, Signal 1.

```matlab
signalbuilder(block,'hidesignal','Signal 1', 'Group 1')
```

Signal 1 is no longer visible in the Signal Builder block.

Make Signal 1 visible again.

```matlab
signalbuilder(block,'shows signal', 'Signal 1', 'Group 1')
```

Create Two Signal Builder Blocks

Create two Signal Builder blocks in new model editor windows.

```matlab
block = signalbuilder([], 'create', [0, 5], {[2 2];[0 2]});
block1 = signalbuilder('untitled/Signal Builder1', 'create', [1, 2], {[1 2];[0 10]});
```

Get a dataset for group 1 of block.

```matlab
ds=signalbuilder(block,'get',1);
```

Get a dataset for group 1 of block1.

```matlab
ds1=signalbuilder(block1,'get',1);
```

Set the dataset for group 1 of block to ds1.

```matlab
signalbuilder(block,'set',1,ds1);
```

Append the original dataset for group 1 of block (ds) to block.

```matlab
signalbuilder(block,'appendgroup',ds);
```
To create a third group in `block`, append `ds1` to the end of the groups in `block`.

```matlab
signalbuilder(block,'appendgroup',ds1);
```

## Input Arguments

**block — Signal Builder block path or handle**

[] (default) | block handle | block name

Signal Builder block handle or name. If you specify [] for this argument, the block has the path 'untitled/Signal Builder'.

Example: 'untitled/Signal Builder'

Example: `block_handle = gcbh`

Data Types: char | string

**time — Time**

row vector | column cell vector | row cell vector | cell matrix

Specify `time` format depending on the block configuration.

If `data` is a cell array and `time` is a vector, the `time` values are duplicated for each element of `data`. Each vector within `time` and `data` must be the same length and have at least two elements. If `time` is a cell array, all elements in a column must have the same initial and final value.

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Time Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 signal, 1 group</td>
<td>Row vector of break points.</td>
</tr>
<tr>
<td>&gt;1 signal, 1 group</td>
<td>Column cell vector where each element corresponds to a separate signal and contains a row vector of points.</td>
</tr>
<tr>
<td>1 signal, &gt;1 group</td>
<td>Row cell vector where each element corresponds to a separate group and contains a row vector of points.</td>
</tr>
<tr>
<td>&gt;1 signal, &gt;1 group</td>
<td>Cell matrix where each element (i, j) corresponds to signal i and group j.</td>
</tr>
</tbody>
</table>
Dependencies

If `signalbuilder` is called for an existing block, the `time` argument must have the same number of signals as the existing block.

Data Types: double

**data — Data**

`row vector | column cell vector | row cell vector | cell matrix`

Specify `data` format depending on the block configuration.

If `data` is a cell array and `time` is a vector, the `time` values are duplicated for each element of `data`. Each vector within `time` and `data` must be the same length and have at least two elements. If `time` is a cell array, all elements in a column must have the same initial and final value.

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Time/Data Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 signal, 1 group</td>
<td>Row vector of break points.</td>
</tr>
<tr>
<td>&gt;1 signal, 1 group</td>
<td>Column cell vector where each element corresponds to a separate signal and contains a row vector of points.</td>
</tr>
<tr>
<td>1 signal, &gt;1 group</td>
<td>Row cell vector where each element corresponds to a separate group and contains a row vector of points.</td>
</tr>
<tr>
<td>&gt;1 signal, &gt;1 group</td>
<td>Cell matrix where each element (i, j) corresponds to signal i and group j.</td>
</tr>
</tbody>
</table>

Data Types: double

**config — Configuration structure name**

`structure name`

Configuration structure containing instructions to display signal group information on the block mask. Set up the structure with one or more of these fields before you print.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Example Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>groupIndex</td>
<td>Scalar specifying index of signal group to print</td>
<td>2</td>
</tr>
<tr>
<td>Field</td>
<td>Description</td>
<td>Example Value</td>
</tr>
<tr>
<td>---------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>---------------</td>
</tr>
<tr>
<td>timeRange</td>
<td>Two-element vector specifying the time range to print (must not exceed the block's time range)</td>
<td>[3 6]</td>
</tr>
<tr>
<td>visibleSignals</td>
<td>Vector specifying index of signals to print</td>
<td>[1 2]</td>
</tr>
<tr>
<td>yLimits</td>
<td>Cell array specifying limits for each signal's y-axis</td>
<td>{[-1 1], [0 1]}</td>
</tr>
<tr>
<td>extent</td>
<td>Two-element vector of the form: [width, height] specifying the dimensions (in pixels) of the area in which to print the signals</td>
<td>[500 300]</td>
</tr>
<tr>
<td>showTitle</td>
<td>Logical value specifying whether to print a title: true (1) prints the title</td>
<td>false</td>
</tr>
</tbody>
</table>

For example, to print just group 2 using a configuration structure, configstruct, set up the structure as follows. You do not need to specify any other fields.

```matlab
configstruct.groupIndex=1
```

Example: configstruct

Data Types: char | string

'create' — Create directive

'create'

Create new Signal Builder block.

Data Types: char | string

signames — Signal names

<table>
<thead>
<tr>
<th>Signal 1 (default)</th>
<th>' '</th>
<th>{}</th>
<th>character vector</th>
<th>cell array of character vectors</th>
</tr>
</thead>
</table>

Signal names, specified as ' ', {}, a character vector, or cell array of character vectors.

If you specify a value of ' ' or {}, the function uses existing signal names for the new groups.

Data Types: char | string
**groupnames — Signal group names**
Group 1 (default) | character vector | cell array of character vectors

Group names, specified as a character vector or cell array of character vectors.
Data Types: char | string

**path — Block path**
[] (default) | full block path

Block path, specified as full block path. To create a Signal Builder block in a new model, `untitled`, with the name Signal Builder, specify `[]`.
Data Types: char | string

**vis — Signal visibility**
matrix

Signal visibility, specified as a matrix. The Signal Builder block displays signals in each group based on the values of the matrix `vis`. This matrix must be the same size as the cell array, `data`. You cannot create Signal Builder blocks in which all signals are invisible. For example, if you set the `vis` parameter for all signals to 0, the first signal is still visible.
Data Types: double

**pos — Block position in model**
[0 0] (default) | [x y right bottom]

Block position in model, specified as `[x y right bottom]`.
Data Types: double

**openui — Open Signal Builder block dialog box**
0 (default) | 1

Open Signal Builder block dialog upon creation by `signalbuilder` function, specified as 0 or 1.
Data Types: double

**openmodel — Open model**
1 (default) | 0

Open model upon creation by `signalbuilder` function, specified as one of:
0 — Load the model but do not open it.
1 — Open the model.

Data Types: double

'appendsignal' — Append signals
'appendsignal' (default)

Append new signals to all signal groups in the Signal Builder block. You can append either the same signals to all groups, or append different signals to different groups. Regardless of which signals you append, append the same number of signals to all the groups. Append signals to all the groups in the block. You cannot append signals to a subset of the groups.

Data Types: char | string

'appendgroup' — Append new signal group
'appendgroup'

Append new signal groups whose number of time and signal elements are the same as existing signal groups in the Signal Builder block.

For the showsignal and hidesignal methods, if you do not specify a value for the group argument, signalbuilder applies the operation to all the signals and groups.

Data Types: char

'set' — Set values
'set'

Set values based on input arguments.

• time and data — Return time and data values.
• ds — Return Simulink.SimulationData.Dataset object(s).

Data Types: char

'get' — Get values
'get'

Get values based on output arguments.

• [time, data] — Return time and data values.
• `[ds, ...dsN]` — Return `Simulink.SimulationData.Dataset` object(s).

  Data Types: `char` | `string`

  `'showsignal'` — Make signal visible
  `'showsignal'`

  Make one or more signals visible in a signal group. If no group is specified, all the signals and groups are visible.

  Data Types: `char` | `string`

  `'hidesignal'` — Hide signal
  `'hidesignal'`

  Hide one or more signals in a signal group. If no group is specified, all the signals and groups are hidden.

  Data Types: `char`

  **ds — Dataset**
  
  `timeseries` data elements | 1-D scalar data value at each time | nonempty data and/or time | empty values

  Dataset of timeseries elements, specified as a `Simulink.SimulationData.Dataset` object.

  Data Types: `double`

  **ds1 ...dsN — Multiple datasets**
  
  vectors of `timeseries` data elements | vectors of 1-D scalar data value at each time | nonempty data and/or time

  One or more datasets of timeseries elements, specified as `Simulink.SimulationData.Dataset` objects.

  Example: `[ds1 ds2 ds3]`

  Data Types: `double`

  **signal — Signal**
  
  signal name | scalar index | array of signal indices

  Signal, specified as a signal name, scalar index, or array of signal indices.
Data Types: char | double | string

**group — Signal group**
group name | scalar index | array of group indices

Signal group, specified as a group name, scalar index, or array of group indices.
Data Types: char | double | string

**index — Group index**
active group index

Active signal group index.
Data Types: double

'activegroup' — Get active group
'activegroup'

Get currently active signal group.
Data Types: char | string

'annotategroup' — Display active signal group name
'on' (default) | 'off'

Display active signal group name on Signal Builder block mask:
- 'on' — Display active signal group names on block mask.
- 'off' — Do not display active signal group names on block mask.

Data Types: char | string

'print' — Print signal group
'print'

Print signal group.
Data Types: char | string

**printArgs — Configure print options**
[] | print function arguments

Configure print options (see print).
To print the entire contents of the Signal Builder block, specify [].

Data Types: char | string

'figure' — Figure
'figure'

Create a figure containing contents of Signal Builder block.

Data Types: char | double | string

Output Arguments

time — Time

row vector | column cell vector | row cell vector | cell matrix

Time for the Signal Builder block, returned as row vector, column cell vector, row cell vector, or cell matrix. For the Signal Builder block, time is the x-coordinate.

time returns in different formats depending on the block configuration.

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Time Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 signal, 1 group</td>
<td>Row vector of break points.</td>
</tr>
<tr>
<td>&gt;1 signal, 1 group</td>
<td>Column cell vector where each element corresponds to a separate signal and contains a row vector of points.</td>
</tr>
<tr>
<td>1 signal, &gt;1 group</td>
<td>Row cell vector where each element corresponds to a separate group and contains a row vector of points.</td>
</tr>
<tr>
<td>&gt;1 signal, &gt;1 group</td>
<td>Cell matrix where each element (i, j) corresponds to signal i and group j.</td>
</tr>
</tbody>
</table>

data — Data

row vector | column cell vector | row cell vector | cell matrix

Data of the Signal Builder block, returned as a one-dimensional array. For the Signal Builder block, time is the y-coordinate.

data takes different formats depending on the block configuration.
### Configuration | Data Format
---|---
1 signal, 1 group | Row vector of break points.
>1 signal, 1 group | Column cell vector where each element corresponds to a separate signal and contains a row vector of points.
1 signal, >1 group | Row cell vector where each element corresponds to a separate group and contains a row vector of points.
>1 signal, >1 group | Cell matrix where each element (i, j) corresponds to signal i and group j.

Datasets must have the same number of elements as the signals in a signal group. Dataset format limitations for the `set`, `append`, and `appendgroup` functions include:

- Elements must be MATLAB timeseries data.
  - Timeseries data and/or time cannot be empty.
- Timeseries data must be of type double.
- Timeseries data must be 1-D (scalar value at each time).

#### signames — Signal names
character vector | cell array of character vectors

Signal names, returned as a character vector or cell array of character vectors.

#### block — Signal Builder block path or handle
block handle

Signal Builder block path or handle, returned by block creation or append commands.

#### ds — Dataset
timeseries data elements | 1-D scalar data value at each time | nonempty data and/or time

Dataset of timeseries elements, returned as a `Simulink.SimulationData.Dataset` object.

#### ds, ...dsN — Multiple datasets
vectors of timeseries data elements | vectors of 1-D scalar data value at each time | nonempty data and/or time
One or more datasets of timeseries elements, returned as Simulink.SimulationData.Dataset objects.

**index — Group index**
active group index

Active signal group index.

**activeGroupLabel — Active group label**
index

Active group label, returned as an index.

**figh — Figure handle**
handle

Figure handle, returned when contents of Signal Builder block are printed to the hidden figure.

## More About

### Interpolating Missing Data Values

When specifying a periodic signal such as a Sine Wave, the signalbuilder function uses linear Lagrangian interpolation to compute data values for time steps that occur between time steps for which the signalbuilder function supplies data. When specifying periodic signals, specify them as a time vector that is defined as multiples of sample time, for example:

```matlab
t = 0.2*[0:49]';
```

## See Also

Signal Builder | Signal Editor | Simulink.SimulationData.Dataset

## Topics

“Load Data with Interchangeable Scenarios”
“Signal Groups”
Introduced in R2007a
signalEditor

Start Signal Editor

Syntax

signalEditor
signalEditor(Name,Value)

Description

signalEditor starts Signal Editor without an associated model.

signalEditor(Name,Value) starts signal Editor using additional options specified by one or more name-value pair arguments.

Specify optional comma-separated pairs of Name, Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside single quotes ('). You can specify the name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

• Model - Model name, specified as a character array, for which Signal Editor is to start. You can specify one model per call to the signalEditor function.

Note Load the model before starting Signal Editor for it.

• DataSource - Data set name, specified as a character array, to be edited. You can specify one data set file per call to the signalEditor function.

Note You can start multiple sessions of Signal Editor for the same model. However, you can associate a data set file with only one Signal Editor at a time. A data set file cannot have multiple Signal Editor sessions associated with it.

Examples
**Start Signal Editor for a Model**

Start the Signal Editor for the model, slexAutotransRootInportsExample.

Load the slexAutotransRootInportsExample model, then start Signal Editor for it.

```matlab
open_system('slexAutotransRootInportsExample')
signalEditor('Model', 'slexAutotransRootInportsExample');
```

**Start Signal Editor to Edit a Data Set File**

Start Signal Editor to edit myFile.mat.

```matlab
signalEditor('DataSource','myFile.mat');
```

**Start Signal Editor and Data Set File for a Model**

Load the slexAutotransRootInportsExample model, then start Signal Editor for the model, and edit myFile.mat.

```matlab
open_system('slexAutotransRootInportsExample')
signalEditor('Model','slexAutotransRootInportsExample','DataSource','myFile.mat');
```

**See Also**

**Topics**

“Create and Edit Signal Data”

**Introduced in R2017b**
sim

Simulate dynamic system

Syntax

simOut = sim(model)
simOut = sim(model,Name,Value)
simOut = sim(model,ParameterStruct)
simOut = sim(model,ConfigSet)
simOut = sim(model,'ReturnWorkspaceOutputs','on')
simOut = sim(simIn)

Description

simOut = sim(model) simulates the specified model using existing model configuration parameters, and returns the result as either a Simulink.SimulationOutput object (single-output format) or as a time vector compatible with Simulink version R2009a or earlier. See “Backward-Compatible Syntax” on page 2-526.

To return simulation results using the single-output format (simulation object), Single simulation output on the Data Import/Export pane of the Configuration Parameters dialog box is selected by default This selection overrides the Dataset format used for signal logging.

To return simulation results using the backward-compatible format (time vector),

simOut = sim(model,Name,Value) simulates the specified model using parameter name-value pairs.

simOut = sim(model,ParameterStruct) simulates the specified model using the parameter values specified in the structure ParameterStruct.

simOut = sim(model,ConfigSet) simulates the specified model using the configuration settings specified in the model configuration set ConfigSet.
simOut = sim(model,'ReturnWorkspaceOutputs','on') simulates the specified model using existing model configuration parameters, and returns the result as a Simulink.SimulationOutput object (single-output format).

simOut = sim(simIn) simulates a model using the inputs specified in the SimulationInput object, simIn. The sim command is also used with an array of SimulationInput objects to run multiple simulations in a series. If simIn is an array of Simulink.SimulationInput objects, output is returned as an array of Simulink.SimulationOutput objects.

Examples

Simulate Model with Default Parameter Settings

Simulate the model vdp using default model configuration parameters.

simOut = sim('vdp','ReturnWorkspaceOutputs','on')

Simulate Model with sim Command-Line Options

Simulate the model, vdp, in Accelerator mode for an absolute tolerance of 1e-5 and save the states in xoutNew and the output in youtNew.

Specify parameters as name-value pairs to the sim command:

simOut = sim('vdp','SimulationMode','accel','AbsTol','1e-5',...  'SaveState','on','StateSaveName','xoutNew',...  'SaveOutput','on','OutputSaveName','youtNew')

simOut =
  Simulink.SimulationOutput:
     xoutNew: [65x2 double]
     youtNew: [65x2 double]
  SimulationMetadata: [1x1 Simulink.SimulationMetadata]
  ErrorMessage: [0x0 char]
Simulate Model with `sim` Command-Line Options in Structure

Simulate the model, `vdp`, in Accelerator mode for an absolute tolerance of `1e-5` and save the states in `xoutNew` and the output in `youtNew`.

Specify parameters using a name-value pairs structure `paramNameValStruct` for the `sim` command:

```matlab
paramNameValStruct.SimulationMode = 'accel';
paramNameValStruct.AbsTol = '1e-5';
paramNameValStruct.SaveState = 'on';
paramNameValStruct.StateSaveName = 'xoutNew';
paramNameValStruct.SaveOutput = 'on';
paramNameValStruct.OutputSaveName = 'youtNew';
simOut = sim('vdp',paramNameValStruct)
```

`simOut` =
```
Simulink.SimulationOutput:
    xoutNew: [65x2 double]
    youtNew: [65x2 double]
    SimulationMetadata: [1x1 Simulink.SimulationMetadata]
    ErrorMessage: [0x0 char]
```

Simulate Model with `sim` Command-Line Options in Configuration Set

Simulate the model, `vdp`, in Accelerator mode for an absolute tolerance of `1e-5` and save the states in `xoutNew` and the output in `youtNew`.

Specify parameters as name-value pairs in configuration set `mdl_cs` for the `sim` command:

```matlab
mdl = 'vdp';
load_system(mdl)
simMode = get_param(mdl, 'SimulationMode');
set_param(mdl, 'SimulationMode', 'accel')
cs = getActiveConfigSet(mdl);
mdl_cs = cs.copy;
set_param(mdl_cs,'AbsTol','1e-5',...    'SaveState','on','StateSaveName','xoutNew',...
simOut = sim(mdl, mdl_cs)
set_param(mdl, 'SimulationMode', simMode)

simOut = Simulink.SimulationOutput:
    xoutNew: [65x2 double]
    youtNew: [65x2 double]
    SimulationMetadata: [1x1 Simulink.SimulationMetadata]
    ErrorMessage: [0x0 char]

For more information, see Simulink.ConfigSetRef.

**Input Arguments**

*model* — Model to simulate
character vector

Name of model to simulate, specified as a character vector.

Example: `simOut = sim('vdp')`

*ParameterStruct* — Structure containing parameter settings
structure

Structure with fields that are the names of the configuration parameters for the simulation. The corresponding values are the parameter values.

Example: `simOut = sim('vdp', paramNameValStruct)`

*ConfigSet* — Configuration set
object

The set of configuration parameters for a model.

Example: `simOut = sim('vdp', mdl_cs)`

*simIn* — SimulationInput object for a model
object
SimulationInput object created by specifying the model name. For more information, see Simulink.SimulationInput.

Example: simIn = Simulink.SimulationInput('vdp')

Use the SimulationInput object to specify Block Parameters, Model Parameters, Variables and External Inputs to a model to be simulated.

Example: simIn = simIn.setBlockParameter('CSTR/Feed Temperature', 'Value', '300'); simIn = simIn.setModelParameter('StartTime', '1'); simIn = simIn.setVariable('FeedTemp0', 320)

**Name-Value Pair Arguments**

Specify optional comma-separated pairs of Name,Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside quotes. You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

Example: 'Solver','ode15s', 'TimeOut', '30' specifies that the model is simulated using the ode15s solver with a maximum simulation time of 30 seconds.

The `sim` command accepts all simulation parameters as name-value pair arguments. See “Model Parameters” on page 6-2 for a list of all simulation parameters.

In addition, the `sim` command accepts the following parameters that are not available on the Configset and `set_param`

**CaptureErrors — Save errors to SimulationMetadata object**

off (default) | on

By default, if an error occurs during simulation, the `sim` command stops and reports the error in the MATLAB Command Window. If you specify 'CaptureErrors', 'on', the `sim` command does not stop, but instead saves any errors to the ErrorDiagnostic structure within the SimulationMetadata object. The error message is saved in the ErrorMessage property of the SimulationOutput object.

This option is useful when running multiple simulations in a loop, so that one simulation error will not stop a script or function from continuing.

If you specify an array of input objects, the `sim` commands runs with CaptureErrors enabled. If an error occurs, the error messages are included in the
Simulink.SimulationMetadata object for the simulation, as well as simulation data up to the point of the failure.

This option is not available for simulation in SIL and PIL modes.

Example: 'CaptureErrors','on'

ConcurrentResolvingToFileSuffix — Append suffix to model name
character vector

(Rapid Accelerator mode only) Appends this suffix character vector to the filename of a model (before the file extension) if:

- The model contains a To File block.
- You call the sim command from parfor.

Example: 'ConcurrentResolvingToFileSuffix','model'

Debug — Run simulation in debug mode
off (default) | on | cmds | cell array

Setting the value of 'Debug' argument to 'on' starts the simulation in debug mode (see “Debugger Graphical User Interface” for more information). The value of this option can be a cell array of commands to be sent to the debugger after it starts.

Example: 'Debug','on'

LoggingFileName — Specify name of MAT-file to log data
out.mat (default) | character vector

Use when you enable the LoggingToFile name-value pair for logging to persistent storage. Specify the destination MAT-file for data logging.

Tip Do not use a file name from one locale in a different locale.

Example: 'LoggingFileName','out.mat'

LoggingToFile — Log simulation data to MAT-file
off (default) | on

Store logged data that uses Dataset format to persistent storage (MAT-file).
Use this feature when logging large amounts of data that can cause memory issues. For details, see “Log Data to Persistent Storage”.

**Tip** To avoid running out of memory when accessing stored data, you can use a reference to access the object stored in the MAT-file. Use a `Simulink.SimulationData.DatasetRef` object to access stored data by reference. Using this object loads signal logging and states data into the model workspace incrementally (signal by signal). Accessing data for other kinds of logging loads all of the data at once.

Example: 'LoggingToFile','on'

**RapidAcceleratorParameterSets** — Pass run-time parameters directly to Rapid Accelerator simulations

structure

(Rapid Accelerator mode only) Returns structure that contains run-time parameters for running Rapid Accelerator simulations in `parfor`.

Example: 'RapidAcceleratorParameterSets',parameterSet(idx)

**RapidAcceleratorUpToDateCheck** — Perform up-to-date check before simulation

on (default) | off

(Rapid Accelerator mode only) Enables/disables up-to-date check. If you set this value to 'off', Simulink does not perform an up-to-date check. It skips the start/stop callbacks in blocks. If you call the `sim` command from `parfor`, set this value to 'off'.

When you set this option to 'off', changes that you make to block parameter values in the model (for example, by using block dialog boxes, by using the `set_param` function, or by changing the values of MATLAB variables) do not affect the simulation. Instead, use `RapidAcceleratorParameterSets` to pass new parameter values directly to the simulation.

Example: 'RapidAcceleratorUpToDateCheck','off'

**SrcWorkspace** — Workspace in which to evaluate MATLAB expressions

base (default) | current | parent

Specifies the workspace in which to evaluate MATLAB expressions defined in the model. Setting `SrcWorkspace` has no effect on a referenced model that executes in Accelerator
mode. Setting SrcWorkspace to current within a parfor loop causes a transparency violation.

Example: 'SrcWorkspace','current'

**TimeOut — Maximum simulation run time**
positive scalar

Specify the time, in seconds, to allow the simulation to run. If you run your model for a period longer than the value of TimeOut, the software issues a warning and stops the simulation. TimeOut refers to the time spent for a simulation.

Example: 'TimeOut',60

**Trace — Enables simulation tracing facilities**
minstep|siminfo|compile

Enables simulation tracing facilities (specify one or more as a comma-separated list):

- 'minstep' specifies that simulation stops when the solution changes so abruptly that the variable-step solvers cannot take a step and satisfy the error tolerances.
- 'siminfo' provides a short summary of the simulation parameters in effect at the start of simulation.

By default, Simulink issues a warning message and continues the simulation.

Example: 'Trace','minstep','Trace','siminfo','Trace','compile'

**Output Arguments**

**simOut — Simulation object containing logged simulation results**
object

Simulink.SimulationOutput object that contains all of the logged simulation results.

All simulation outputs (logged time, states, and signals) are returned in a single Simulink.SimulationOutput object. You define the model time, states, and output that is logged using the **Data Import/Export** pane of the **Model Configuration Parameters** dialog box. You can log signals using blocks such as the To Workspace and Scope blocks. The **Viewers and Generators Manager** can directly log signals.
Note  The output of the sim command always returns to SimOut, the single simulation output object. The simulation output object in turn, is returned to the workspace.

More About

Backward-Compatible Syntax

Starting with R2009b, the sim command was enhanced to provide greater compatibility with parallel computing. The improved single-output format saves all simulation results to a single object, simplifying the management of output variables.

For backward compatibility with R2009a or earlier releases, use the backward-compatible syntax:

\[ [T, X, Y] = \text{sim}('model', \text{Timespan}, \text{Options}, \text{UT}) \]
\[ [T, X, Y_1, \ldots, Y_n] = \text{sim}('model', \text{Timespan}, \text{Options}, \text{UT}) \]

If you specify only the model argument, Simulink automatically saves the time, state, and output to the specified output arguments.

If you do not specify any output arguments, Simulink determines what data to log based on the settings for the Configuration Parameters > Data Import/Export pane. Simulink stores the simulation output either in the current workspace or in the variable ans, based on the setting for Save simulation output as a single object parameter.

Backward-Compatible Syntax Input and Output Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( T )</td>
<td>The time vector returned.</td>
</tr>
<tr>
<td>( X )</td>
<td>The state returned in matrix or structure format. The state matrix contains continuous states followed by discrete states.</td>
</tr>
<tr>
<td>( Y )</td>
<td>The output returned in matrix or structure format. For block diagram models, this variable contains all root-level blocks.</td>
</tr>
<tr>
<td>( Y_1, \ldots, Y_n )</td>
<td>The outports, which can only be specified for diagram models. Here, ( n ) must be the number of root-level blocks. Each outport will be returned in the ( Y_1, \ldots, Y_n ) variables.</td>
</tr>
<tr>
<td>'model'</td>
<td>The name of the model to simulate.</td>
</tr>
</tbody>
</table>
Simulink requires only the `model` argument. Simulink takes all defaults from the block diagram, including unspecified options. If you specify any optional arguments, your specified settings override the settings in the block diagram.

Specifying an input argument of `sim` as the empty matrix, `[]`, causes Simulink to use the default for that argument.

**Backward-Compatible Syntax Examples**

This command simulates the Van der Pol equations, using the `vdp` model. The command uses all default parameters.

```
[t,x,y] = sim('vdp')
```

This command simulates the Van der Pol equations, using the parameter values associated with the `vdp` model.

```
[t,x,y] = sim('vdp', [0 10]);
```

**Tips**

- Parameters specified using the `sim` command override the values defined in the *Model Configuration Parameters* dialog box. The software restores the original configuration values at the end of simulation.

- In the case of a model with a model reference block, the parameter specifications are applied to the top model.

- When simulating a model with infinite stop time, to stop the simulation, you must press `Ctrl+C`. `Ctrl+C` breaks the simulation and the simulation results are not saved in the MATLAB workspace.
To specify the time span for a simulation, you must specify the **StartTime** and **StopTime** parameters.

To log the model time, states, or outputs, use the **Data Import/Export** pane of the Model Configuration Parameters dialog box.

To log signals, either use a block such as the To Workspace block or the Scope block, or use the Viewers and Generators Manager to log results directly.

To get a list of simulation parameters for the model `vdp`, in the MATLAB Command Window, enter:

```matlab
cfgSet = getActiveConfigSet('vdp')
cfgSetNames = get_param(cfgSet, 'ObjectParameters')
```

This command lists several object parameters, including simulation parameters such as 'StartTime', 'SaveTime', 'SaveState', 'SaveOutput', and 'SignalLogging'.

### See Also

**Functions**
- parsim
- sldebug

**Classes**
- ConfigSet
- Simulink.ConfigSetRef
- Simulink.SimulationInput
- Simulink.SimulationOutput

**Blocks**
- Scope
- To Workspace

**Topics**
- “Run Simulations Programmatically”
- “Run Multiple Simulations”
- “About Configuration Sets”
- “Configuration Parameters Dialog Box Overview”
- “Log Data to Persistent Storage”
- “Model Parameters” on page 6-2

**Introduced before R2006a**
simplot

Redirects to the Simulation Data Inspector

**Note** simplot will be removed in a future release. Use the Simulation Data Inspector instead.

**Syntax**

simplot

**Description**

simplot redirects to the Simulation Data Inspector and returns empty handles. This function is no longer supported and has been replaced by the Simulation Data Inspector. Use the **Simulation Data Inspector** button in the Simulink Editor to capture simulation output in the Simulation Data Inspector. Programmatically, use the function Simulink.sdi.view instead.

**See Also**

Simulink.sdi.view
Simulation Manager

Monitor multiple simulations and visualize the simulation data

Description

Simulation Manager allows you to monitor the status of multiple simulations. Using this tool, you can:

- View the progress of the simulations in a high-level grid view or a detailed list view.
- Find the simulations that error out and view diagnostics for each simulation.
- Visualize simulation data to analyze the trends across the simulations.
- Select a simulation run and open the model in Simulink, with all of the simulation settings applied to the model.
- View simulation results in the Simulation Data Inspector.
- Abort simulations.

Open the Simulation Manager

- parsim and sim commands — Set the argument ShowSimulationManager to on with Parallel Computing Toolbox. The parsim and sim commands use a Simulink.SimulationInput object to run simulations. For more information, see Simulink.SimulationInput.

\[
\text{out} = \text{parsim}(\text{in}, '\text{ShowSimulationManager}', 'on')
\]

\[
\text{out} = \text{sim}(\text{in}, '\text{ShowSimulationManager}', 'on')
\]

- To open a saved Simulation Manager session, navigate to the folder where the file is saved and double click the .mldatx Simulation Manager file.

\[
\text{open} \ \text{filename.mldatx}
\]
Examples

Open Simulation Manager

Open the model sldemo_suspn_3dof and create a set of sweep values.

mdl = 'sldemo_suspn_3dof';
open_system(mdl);
Cf_sweep = Cf*(0.05:0.1:0.95);
numSims = length(Cf_sweep);

To modify the block parameter Road-Suspension Interaction with the sweep values, create an array of Simulink.SimulationInput objects.

in(1:numSims) = Simulink.SimulationInput(mdl);
for i = 1:numSims
  in(i) = setBlockParameter(in(i), [mdl '/Road-Suspension Interaction'], 'Cf', num2str(Cf_sweep(i)));
end

Run multiple simulations and open the Simulation Manager.

out = parsim(in, 'ShowSimulationManager', 'on')

In the absence of Parallel Computing Toolbox, the simulations run in serial.
Using Simulation Manager

Once you run the `parsim` command, the Simulation Manager UI opens up as follows:

![Simulation Manager UI](image)

**Plot results**

In the **Figure 1** tab, you can view the default scatter plot where the parameter \( C_f \) is plotted against itself. Use the **Plot Properties** to edit the plot. Add grid lines for \( X \) and \( Y \) by checking the corresponding boxes. You can also change the data that is displayed on X-
and Y-axes and the Marker Color. Change the data on y-axis by selecting ScopeData(1) from the Data drop-down. Set the Marker Color Data field to RunID.

To add another figure to the Simulation Manager, click surf or scatter plot and add plots as desired. In this example, click surf plot, located in the Results section of the toolstrip. Drag the Figure 2 tab to view the two plots together.

Plots in the Simulation Manager support numeric arrays and datasets.
Adding multiple plots to the Simulation Manager enables you to observe the simulation data in different ways. Using the results, you can analyze and study the design space of the parameters and their behavior.

You can modify the configuration of the plot layouts by clicking the down arrow on the left corner of the plot.

- **Tile All** — Choose a layout for multiple plots.
- **Tab Position** — Select where to display the figure tab.

**View the Status**

To view the status of the simulations in list and grid views, click the **Simulations** tab next to **Figure 1**. Toggle between the **List** and **Grid** in the **Views** section of the toolstrip. The list view shows a tabular view of the simulations with certain details, such as status, parameters and timing information. For large number of simulations, choose grid view for a compact view of the status of the simulations.
The status bar at the bottom of the window shows the progress of the simulations.

You can view all the multiple simulations in a detailed list view. This view gives you an option to add or delete columns. Use the button to choose which columns to display. You can also sort the columns based on your preference.

Click a specific run to view more details about it. The simulation details appear at the bottom of the window.
To show or hide the details of the selected simulation, toggle the **Simulation Details** button.

**Open Selected** button, allows you to open the model with the specifications of the selected run.

You can view the results of one or more runs in the Simulation Data Inspector by clicking the **Show Results** button. Clicking Show Results creates a Simulation Data Inspector run from the `Simulink.SimulationOutput` object and is displayed in Simulation Data Inspector.

To abort the job while the simulations are running, you can use the **Stop Job** button.

**Save the Session**

To save the Simulation Manager session, click the **Save** button on the toolstrip. The saved session contains all the simulation data and the plot configurations and layout. To reopen the saved Simulation Manager session, navigate to the folder where the file is saved and double-click the `.mldatx` Simulation Manager file.

The **Reuse Window** button enables you to reuse the plot layout and configurations for new multiple simulations run with a different set of data. If the button is toggled off, a new window opens with each set of simulations. If the button is toggled on, the new simulations reuse the existing window preserving all plot configurations and layouts.

To reopen a saved file, use the **Open** button, navigate to the file location and double click the file.
See Also

Functions
Simulink.SimulationInput | applyToModel | parsim | setBlockParameter | setExternalInput | setInitialState | setModelParameter | setPostSimFcn | setPreSimFcn | setVariable | sim | validate

Topics
“Run Parallel Simulations Using parsim”
“Run Multiple Simulations”
“Analyze Results Using Simulation Manager”
“Run Parallel Simulations”
simulink

Open Simulink Start Page

Syntax

simulink

Description

simulink opens the Simulink Start Page. From the Start Page, choose a model or project template, or browse the examples. See “Create and Open Models”.

The behavior of the simulink function changed in R2016a. Formerly it opened the Simulink Library Browser and loaded the Simulink block library.

- To open the Library Browser, use slLibraryBrowser.
- To only load the Simulink block library, use this command:
  
  load_system simulink

- To start Simulink without opening the Library Browser or Start Page, use start_simulink, which is faster.

See Also

slLibraryBrowser | start_simulink

Topics

“Create and Open Models”
“Modeling”

Introduced before R2006a
simulinkproject

(Not recommended) Open project and get project object

**Note** simulinkproject is not recommended. Use currentProject or openProject instead. For more information, see “Compatibility Considerations”.

**Syntax**

```matlab
simulinkproject
simulinkproject(projectPath)

proj = simulinkproject
proj = simulinkproject(projectPath)
```

**Description**

`simulinkproject` opens the project or brings focus to the tool if it is already open. After you open the tool, you can create projects or access recent projects using the **Project** tab.

`simulinkproject(projectPath)` opens the project specified by any file or folder under the project root in `projectPath` and gives focus to the project.

`proj = simulinkproject` returns a project object `proj` you can use to manipulate the project at the command line. You need to get a project object before you can use any of the other project functions.

`proj = simulinkproject(projectPath)` opens the project specified by `projectPath` and returns a project object.

**Examples**
**Open Project Tool**

Open the Project Tool.

`simulinkproject`

**Open a Project**

Specify either the `.prj` file path or the folder that contains your `.SimulinkProject` folder and `.prj` file. The project opens and brings focus to the project.

`simulinkproject('C:/projects/project1/')`

**Open a Project and Get a Project Object**

Open a specified project and get a project object to manipulate the project at the command line. To avoid your startup script opening windows that take focus away from the MATLAB Desktop, use `start_simulink` instead of the `simulink` function, and use `simulinkproject` with an output argument instead of `uiopen`. If you use `uiopen(myproject.prj)` this calls `simulinkproject` with no output argument and gives focus to the project.

`start_simulink`

`proj = simulinkproject('C:/projects/project1/myproject.prj')`

**Get Airframe Example Project**

Open the Airframe project and create a project object to manipulate and explore the project at the command line.

`sldemo_slproject_airframe`

`proj = simulinkproject`

`proj =`

`ProjectManager with properties:`

```
Name: 'Simulink Project Airframe Example'
Information: [1x1 slproject.Information]
```
Find Project Commands

Find out what you can do with your project.

\texttt{methods(proj)}

Methods for class \texttt{slproject.ProjectManager}:

\texttt{addFile}
\texttt{addFolderIncludingChildFiles}
\texttt{addPath}
\texttt{addReference}
\texttt{addShortcut}
\texttt{addShutdownFile}
\texttt{addStartupFile}
\texttt{close}
\texttt{createCategory}
\texttt{export}
\texttt{findCategory}
\texttt{findFile}
\texttt{isLoaded}
\texttt{listModifiedFiles}
\texttt{listRequiredFiles}
\texttt{refreshSourceControl}
\texttt{reload}
\texttt{removeCategory}
\texttt{removeFile}
\texttt{removePath}
\texttt{removeReference}
\texttt{removeShortcut}
removeShutdownFile
RemoveStartupFile

**Examine Project Properties Programmatically**

After you get a project object using the `simulinkproject` function, you can examine project properties.

Examine the project files.

```matlab
files = proj.Files
```

```matlab
files =

1x31 ProjectFile array with properties:

Path
Labels
Revision
SourceControlStatus

Use indexing to access files in this list. The following command gets file number 14. Each file has properties describing its path, attached labels, and source control information.

```matlab
proj.Files(15)
```

```matlab
ans =

ProjectFile with properties:

Path: 'C:\slexamples\airframe24\models\DigitalControl.slx'
Labels: [1x1 slproject.Label]
Revision: '2'
SourceControlStatus: Unmodified

Examine the labels of the file.

```matlab
proj.Files(15).Labels
```

```matlab
ans =

Label with properties:
Get a particular file by name.

```matlab
code = findFile(proj, 'models/AnalogControl.slx')
code =
    ProjectFile with properties:
    Path: 'C:slexamples\airframe24\models\AnalogControl.slx'
        Labels: [1x1 slproject.Label]
        Revision: '2'
        SourceControlStatus: Unmodified
```

Find out what you can do with the file.

```matlab
methods(code)
```

Methods for class slproject.ProjectFile:

- addLabel
- findLabel
- removeLabel

Update the file dependencies.

```matlab
update(proj.Dependencies)
```

The project runs a dependency analysis to update the known dependencies between project files.

For more information on working with project files, including modified files and dependencies, see “Automate Simulink Project Tasks Using Scripts”.

Input Arguments

- **projectPath** — Full path to project file or folder
  character vector

  Full path to project .prj file, or the path to the project root folder, or any subfolder or file under your project root, specified as a character vector.
Example: 'C:/projects/project1/myProject.prj'
Example: 'C:/projects/project1/'

**Output Arguments**

**proj — Project**

project object

Project, returned as a project object. Use the project object to manipulate the currently open project at the command line.

Properties of **proj** output argument.

<table>
<thead>
<tr>
<th>Project Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Categories</td>
<td>Categories of project labels</td>
</tr>
<tr>
<td>Dependencies</td>
<td>Dependencies between project files in a MATLAB digraph object</td>
</tr>
<tr>
<td>Files</td>
<td>Paths and names of project files</td>
</tr>
<tr>
<td>Information</td>
<td>Information about the project such as the description, source control</td>
</tr>
<tr>
<td></td>
<td>integration, repository location, and whether it is a top-level project</td>
</tr>
<tr>
<td>Name</td>
<td>Project name</td>
</tr>
<tr>
<td>ProjectPath</td>
<td>Folders that the project puts on the MATLAB path</td>
</tr>
<tr>
<td>ProjectReferences</td>
<td>Folders that contain referenced projects. Contains read-only project objects</td>
</tr>
<tr>
<td></td>
<td>for referenced projects.</td>
</tr>
<tr>
<td>RootFolder</td>
<td>Full path to project root folder</td>
</tr>
<tr>
<td>Shortcuts</td>
<td>An array of the shortcuts in this project</td>
</tr>
<tr>
<td>ShutdownFiles</td>
<td>An array of the shutdown files in this project</td>
</tr>
<tr>
<td>StartupFiles</td>
<td>An array of the startup files in this project</td>
</tr>
</tbody>
</table>
**Tips**

Alternatively, you can use `slproject.loadProject` to load a project, and `slproject.getProjects` to get a project object. Use `simulinkproject` to open projects and explore projects interactively. Use `slproject.getProjects` and `slproject.loadProject` for project automation scripts.

**Compatibility Considerations**

**Simulink project API is not recommended**

*Not recommended starting in R2019a*

Starting in R2019a, instead of `simulinkproject` and related functions, use the `currentProject` or `openProject` functions in MATLAB. The Simulink project API will continue to be supported, but, after R2019a, new features might be available only if you use the new MATLAB project API. There are no plans to remove the Simulink project API at this time.

The new MATLAB project API is part of the R2019a functionality enabling you to use projects in MATLAB, with or without Simulink. You can now share projects with users who do not have Simulink.

**See Also**

**Functions**

`addFile` | `addFolderIncludingChildFiles` | `addLabel` | `addPath` | `addReference` | `addShortcut` | `createCategory` | `currentProject` | `findFile` | `findLabel` | `listModifiedFiles` | `refreshSourceControl` | `removeFile` | `slproject.create` | `slproject.getProjects` | `slproject.loadProject` | `start_simulink`

**Topics**

“Automate Simulink Project Tasks Using Scripts”
“Create a New Project From a Folder”
“Open Recent Projects”
“Clone Git Repository or Check Out SVN Repository”
“What Are Projects?”
Introduced in R2012a
Simulink.allBlockDiagrams

Find loaded Simulink models and libraries

Syntax

bd = Simulink.allBlockDiagrams()
bd = Simulink.allBlockDiagrams(type)

Description

bd = Simulink.allBlockDiagrams() returns all loaded block diagrams, including models and libraries.

bd = Simulink.allBlockDiagrams(type) returns either models or libraries.

Examples

Find Loaded Models

Find all loaded models in the current Simulink session, excluding libraries. The example shows a result from a typical session.

Simulink.allBlockDiagrams('model')

ans =

237.0001
56.0001
2.0001
Get Names of Loaded Block Diagrams

Find all loaded models in the current Simulink session and return results as names. Use Simulink.allBlockDiagrams with get_param to get the names. The example shows a result from a typical session and includes loaded libraries and models.

```
get_param(Simulink.allBlockDiagrams(),'Name')
```

```
ans =

5×1 cell array

    {'simulink_extras' }
    {'simulink'      }     }
    {'sldemo_fuelsys' }
    {'f14'            }     }
    {'vdp'            }
```

Get Loaded Block Diagrams Based on Parameter

Find all loaded models in the current Simulink session whose 'Dirty' parameter is 'on'.

```
bds = Simulink.allBlockDiagrams();
dirtyBds = bds(strcmp(get_param(bds,'Dirty'),'on'));
```

Input Arguments

- **type** — Type of block diagram whose blocks to return
  
  *'model'* | *'library'*

  Type of block diagram whose blocks to return, specified as *'model'* or *'library'*.

Output Arguments

- **bd** — Loaded block diagrams
  
  array of handles

  Loaded block diagrams, returned as an array of handles.
See Also
Simulink.FindOptions | Simulink.findBlocks | Simulink.findBlocksOfType

Introduced in R2018a
Simulink.architecture.add

Add tasks or triggers to selected architecture of model

Syntax

Simulink.architecture.add(Type,Object)

Description

Simulink.architecture.add(Type,Object) adds the new task or trigger Object of the specified Type to a model.

Examples

Add periodic trigger

Add a task, MyTask1, to the software node MulticoreProcessor of the selected architecture of the slexMulticoreExample model.

slexMulticoreExample;
Simulink.architecture.add('Task','slexMulticoreExample/MulticoreProcessor/Core2/MyTask1');

Input Arguments

Type — Object type

'PeriodicTrigger' | 'AperiodicTrigger' | 'Task'

Object type that identifies the kind of trigger or task to add, specified as a 'PeriodicTrigger', 'AperiodicTrigger', or 'Task'.

• 'PeriodicTrigger'

    Adds a periodic trigger to the architecture. Set the properties of the trigger with the Simulink.architecture.set_param function.
• 'AperiodicTrigger'

  Adds an aperiodic trigger to the architecture. Set the properties of the trigger with the Simulink.architecture.set_param function.

• 'Task'

  Adds a task to the architecture. Set the properties of the task with the Simulink.architecture.set_param function.

**Object — Trigger or task object identifier**

character vector

Trigger or task object identifier to add to architecture, specified as a character vector. Example: 'slexMulticoreExample/MulticoreProcessor/Core2/MyTask1'

Data Types: char

**See Also**

Simulink.architecture.delete | Simulink.architecture.find_system | Simulink.architecture.get_param | Simulink.architecture.importAndSelect | Simulink.architecture.profile | Simulink.architecture.register | Simulink.architecture.set_param

**Introduced in R2014a**
Simulink.architecture.config

Create or convert configuration for concurrent execution

Syntax

Simulink.architecture.config(model,'Convert')
Simulink.architecture.config(model,'Add')
Simulink.architecture.config(model,'OpenDialog')

Description

Simulink.architecture.config(model,'Convert') converts the active configuration set in the specified model to one for concurrent execution.

Simulink.architecture.config(model,'Add') adds and activates a new configuration set for concurrent execution.

Simulink.architecture.config(model,'OpenDialog') opens the Concurrent Execution dialog box for a model configuration.

Examples

Convert existing configuration set

Convert existing configuration set for concurrent execution in the model vdp.

vdp;
Simulink.architecture.config('vdp','Convert');

Add new configuration set

Add a new configuration set (copied from the existing configuration set) for concurrent execution in the model vdp.
vdp; 
Simulink.architecture.config('vdp','Add');

**Open Concurrent Execution dialog box**

Open the Concurrent Execution dialog box in the model slexMulticoreExample.

slexMulticoreExample; 
Simulink.architecture.config('slexMulticoreExample','OpenDialog');

**Input Arguments**

`model` — Model name
character vector

Model name whose configuration set you want to convert or add to, specified as a character vector.

Example:

Data Types: char

**See Also**

Simulink.architecture.add | Simulink.architecture.profile | Simulink.architecture.set_param

Introduced in R2014a
signalBuilderToSignalEditor

Import signal data and properties from Signal Builder block to Signal Editor block

Syntax

signal_editor = signalBuilderToSignalEditor(signal_builder, Name, Value)
[signal_editor, sorted_group_index, sorted_group_names] = signalBuilderToSignalEditor(signal_builder, Name, Value)

Description

signal_editor = signalBuilderToSignalEditor(signal_builder, Name, Value) imports signal data and properties from Signal Builder block to the Signal Editor block. This function adds a Signal Editor block to the current model using the signal data and properties from the Signal Builder block.

During the port, the signalBuilderToSignalEditor function:

- Unicode group names are prefixed with Test_Case_.
- Group names are converted to valid MATLAB variable names with leading and trailing white spaces removed.
- Group names are converted with trailing underscores removed.
- Remaining numbers maintained in the scenario name.
- Orders signal groups alphabetically.
- Creates unique group names from existing names following MATLAB conventions.

The signalBuilderToSignalEditor function does not support

- Models that contain test case parameters. You can successfully port data from the Signal Builder block, but you cannot initialize parameters with the Signal Editor block in test harnesses generated by Simulink Design Verifier.
- Models that contain Signal Builder signal generators created from the **Create and Connect Generator** context menu.
[signal_editor,sorted_group_index,sorted_group_names] = signalBuilderToSignalEditor(signal_builder,Name,Value) outputs vectors containing the signal groups and group names.

**Examples**

**Replace Signal Builder Block with Signal Editor Block**

This example shows how to replace an existing Signal Builder block with a Signal Editor block. To store signals from Signal Builder, the example uses RoadProfiles.mat.

```matlab
model = 'ex_replace_signalbuilder';
open_system(model);
sbBlockH = [model '/Road Profiles'];
seBlockH = signalBuilderToSignalEditor(sbBlockH,...
'Replace',true,'FileName','RoadProfiles.mat');
```

**Input Arguments**

`signal_builder` — Signal Builder block to replace

- current model (default) | scalar

Signal Builder block to replace, specified as a scalar.

Data Types: char | string
**Name-Value Pair Arguments**

Specify optional comma-separated pairs of Name, Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside quotes. You can specify several name and value pair arguments in any order as Name1, Value1,...,NameN, ValueN.

Example: 'FileName','RoadProfiles.mat'

**FileName — MAT-file that stores signals**

'dataset.mat' (default) | scalar

MAT-file that stores signals and properties, specified as a scalar. Do not use a file name from one locale in a different locale. When using the block on multiple platforms, consider specifying just the MAT-file name and having the MAT-file be on the MATLAB path.

Data Types: char | string

**Replace — Replace Signal Builder block with Signal Editor block**

false (default) | true

Replace Signal Builder block with Signal Editor block, specified as true or false.

Data Types: logical

**Output Arguments**

**signal_editor — Signal Editor block handle**

scalar

Signal Editor block handle, specified as a scalar.

**sorted_group_index — List of Signal Builder group indices**

vector

List of Signal Builder group indices, specified as a vector and ordered as they will appear in the Signal Editor.

**sorted_group_names — List of Signal Builder group names**

cell array

Signal Editor group names, specified as a cell array of vectors, in alphabetical order.
The names are unique valid MATLAB variable names generated from the Signal Builder group names.

See Also
Signal Editor | Signal Builder | signalEditor | signalbuilder

Introduced in R2018a
**Simulink.architecture.delete**

Delete triggers and tasks from selected architecture of model

**Syntax**

```matlab
Simulink.architecture.delete(Object)
```

**Description**

`Simulink.architecture.delete(Object)` deletes the specified object trigger or task.

**Examples**

**Delete task Plant**

Delete the task Task3 from the Core2 periodic trigger of the MulticoreProcessor software node of the selected architecture of the slexMulticoreExample model.

```matlab
slexMulticoreExample
Simulink.architecture.delete('slexMulticoreExample/MulticoreProcessor/Core2/Task3')
```

**Input Arguments**

- **Object** — Object to delete, specified as a character vector
  - character vector

  Object to be deleted. Possible objects are:
  
  - Periodic trigger
**Note** You cannot delete the last periodic trigger. The software node must contain at least one periodic trigger.

- Aperiodic trigger
- Task

Example: `[bdroot '/MulticoreProcessor/Core2/Task3']`

Data Types: `char`

**See Also**

`Simulink.architecture.add` | `Simulink.architecture.find_system` | `Simulink.architecture.get_param` | `Simulink.architecture.importAndSelect` | `Simulink.architecture.profile` | `Simulink.architecture.register`

**Introduced in R2014a**
Simulink.architecture.find_system

Find objects under architecture object

Syntax

object = Simulink.architecture.find_system(RootObject)

object = Simulink.architecture.find_system(RootObject,ParamName, ParamValue)

Description

object = Simulink.architecture.find_system(RootObject) looks for all objects under RootObject.

object = Simulink.architecture.find_system(RootObject,ParamName, ParamValue) returns the object in RootObject whose parameter ParamName has the value ParamValue. Parameter name and value character vectors are case-sensitive.

Examples

Look for all objects

To find all the objects in slexMulticoreExample:

slexMulticoreExample

 \texttt{t = Simulink.architecture.find_system('slexMulticoreExample')} 

\texttt{t =

 'slexMulticoreExample'
 'slexMulticoreExample/MulticoreProcessor'
 'slexMulticoreExample/MulticoreProcessor/Core1'
 'slexMulticoreExample/MulticoreProcessor/Core1/Task1'
 'slexMulticoreExample/MulticoreProcessor/Core1/Task2'}
Look for all tasks

To find all the tasks in slexMulticoreExample:

```matlab
slexMulticoreExample
t = Simulink.architecture.find_system('slexMulticoreExample','Type','Task')
t =

'slexMulticoreExample/MulticoreProcessor/Core1/Task1'
'slexMulticoreExample/MulticoreProcessor/Core1/Task2'
'slexMulticoreExample/MulticoreProcessor/Core2/Task3'
'slexMulticoreExample/MulticoreProcessor/Core2/Task4'
```

Input Arguments

**RootObject — Object to search**
character vector

Object to search for parameter value, specified as a character vector giving the object full path name. Possible objects are:

- Model
- Software node
- Hardware node
- Periodic trigger
- Aperiodic trigger
- Task

Example:`'slexMulticoreExample'`

**ParamName — Name of parameter to find**
character vector | scalar | vector

Name of the parameter to find. Possible values are:
• 'Name'
• 'Type'
• 'ClockFrequency'
• 'Color'
• 'Period'
• 'EventHandlerType'
• 'SignalNumber'
• 'EventName'

Example: 'EventName'

**ParamValue — Parameter value to find**

class name | scalar | vector

Parameter value to find, specified as a character vector, a scalar, or a vector.

Example: 'ERTDefaultEvent'

**See Also**

Simulink.architecture.add | Simulink.architecture.delete |
Simulink.architecture.importAndSelect | Simulink.architecture.profile |
Simulink.architecture.register | Simulink.architecture.set_param

**Introduced in R2014a**
Simulink.architecture.get_param

Get configuration parameters of architecture objects

Syntax

ParamValue = Simulink.architecture.get_param(Object,ParamName)

Description

ParamValue = Simulink.architecture.get_param(Object,ParamName) returns
the value of the specified parameter for the object, Object. ParamName is case-sensitive.

Examples

Get period

Get the period of task Task3 of trigger Core2 of software node MulticoreProcessor of the
selected architecture for the model slexMulticoreExample.

slexMulticoreExample;
p = Simulink.architecture.get_param('slexMulticoreExample/MulticoreProcessor/Core2/Task3','Period')

p =
0.2

Input Arguments

Object — Object whose parameter value to return
character vector

Object whose parameter value to return, specified as a character vector giving the object
full path name. Possible objects are:
• Software node
• Hardware node
• Periodic trigger
• Aperiodic trigger
• Task

**ParamName** — **Parameter whose value to return**

character vector

Name of a parameter for which `Simulink.architecture.get_param` returns a value.

The following are the possible `ParamName` values:

For a model:
- 'ArchitectureName'
- 'Type'

For a software node:
- 'Name'
- 'Type'

For a hardware node
- 'Name'
- 'ClockFrequency'
- 'Color'
- 'Type'

For a periodic trigger:
- 'Name'
- 'Period'
- 'Color'
- 'Type'

For an aperiodic trigger:
• 'Name'
• 'Color'
• 'EventHandlerType'
• 'SignalNumber'
• 'EventName'
• 'Type'

For a task:

• 'Name'
• 'Period'
• 'Color'
• 'Type'

See Also
Simulink.architecture.add | Simulink.architecture.delete |
Simulink.architecture.find_system |
Simulink.architecture.importAndSelect | Simulink.architecture.profile |
Simulink.architecture.register | Simulink.architecture.set_param

Introduced in R2014a
Simulink.architecture.importAndSelect

Import and select target architecture for concurrent execution environment for model

Syntax

Simulink.architecture.importAndSelect(model, Architecture)

Simulink.architecture.importAndSelect(model, CustomArchitectureDescriptionFile)

Description

Simulink.architecture.importAndSelect(model, Architecture) imports and selects the built-in target architecture for the concurrent execution environment for the model.

Importing and selecting target architectures requires that the associated support packages or hardware is installed on your computer.

Simulink.architecture.importAndSelect(model, CustomArchitectureDescriptionFile) imports and selects the architecture from an XML-based architecture description file.

Importing and selecting target architectures requires that the associated support packages or hardware is installed on your computer.

Examples

Import and select a different architecture

Import and select the sample architecture to the model slexMulticoreExample.
slexMulticoreExample
Simulink.architecture.importAndSelect('slexMulticoreExample','Sample Architecture')

**Import and select a custom architecture**

Import and select the custom architecture defined in the XML file `custom_arch.xml`. This example requires you to create a `custom_arch.xml` file first.

slexMulticoreExample
Simulink.architecture.importAndSelect('slexMulticoreExample','custom_arch.xml')

**Input Arguments**

**model** — Model
character vector

Model to import architecture to, specified as a character vector.

Data Types: char

**Architecture** — Target architecture name
character vector

Target architecture name to import into the concurrent execution environment for the model. Possible target names are:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'Multicore'</td>
<td>Single CPU with multiple cores</td>
</tr>
<tr>
<td>'Sample Architecture'</td>
<td>Example architecture consisting of single CPU with multiple cores and two FPGAs. You can use this architecture to model for concurrent execution.</td>
</tr>
<tr>
<td>'Simulink Real-Time'</td>
<td>Simulink Real-Time™ target</td>
</tr>
<tr>
<td>'Xilinx Zynq ZC702 evaluation kit'</td>
<td>Xilinx® Zynq® ZC702 evaluation kit target</td>
</tr>
<tr>
<td>'Xilinx Zynq ZC706 evaluation kit'</td>
<td>Xilinx Zynq ZC706 evaluation kit target</td>
</tr>
<tr>
<td>Property</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>'Xilinx Zynq Zedboard'</td>
<td>Xilinx Zynq ZedBoard™ target</td>
</tr>
</tbody>
</table>

Data Types: char

**CustomArchitectureDescriptionFile — Custom target architecture file**

XML file

Custom target architecture file name, in XML format, that describes a custom target for the concurrent execution environment for the model, specified as a character vector giving the XML file name.

Example: custom_arch.xml

**See Also**

Simulink.architecture.add | Simulink.architecture.delete |
Simulink.architecture.find_system | Simulink.architecture.profile |
Simulink.architecture.register | Simulink.architecture.set_param

**Topics**

“Define a Custom Architecture File”

**Introduced in R2014a**
Simulink.architecture.profile

Generate profile report for model configured for concurrent execution

Syntax

Simulink.architecture.profile(model)
Simulink.architecture.profile(model,numSamples)

Description

Simulink.architecture.profile(model) generates a profile report for a model configured for concurrent execution. Subsequent calls to the command for the same model name overwrite the existing profile report.

Simulink.architecture.profile(model,numSamples) specifies the number of samples to generate a profile report.

Examples

Generate profile report

Generate profile report for the model slexMulticoreExample.

slexMulticoreExample;
Simulink.architecture.profile('slexMulticoreExample');

The command creates the file slexMulticoreExample_ProfileReport.html in the current folder and opens it.

Generate profile report for 120 time steps

Generate profile report for the model slexMulticoreExample with data for 120 time steps.
slexMulticoreExample;
Simulink.architecture.profile('slexMulticoreExample',120);

The command creates the file slexMulticoreExample_ProfileReport.html in the current folder.

**Input Arguments**

**model — Model to profile**
character vector

Model to profile, specified as a character vector. Specify a model that is configured for concurrent execution.

Data Types: char

**numSamples — Number of time steps**
100 (default) | real, positive integer

Number of time steps, specified as a real, positive integer. This value determines the number of steps to collect data for in the profiled model.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

**See Also**

Simulink.architecture.add | Simulink.architecture.delete | Simulink.architecture.find_system | Simulink.architecture.get_param | Simulink.architecture.importAndSelect | Simulink.architecture.register | Simulink.architecture.set_param

**Topics**

“Profile and Evaluate Explicitly Partitioned Models on a Desktop”

**Introduced in R2014a**
Simulink.architecture.register

Add custom target architecture to concurrent execution target architecture selector

Syntax

Simulink.architecture.register(CustomArchFile)

Description

Simulink.architecture.register(CustomArchFile) adds an XML-format custom target architecture file CustomArchFile to the concurrent execution target architecture selector. To access this selector, click the Select button on the Concurrent Execution pane of the Concurrent Execution dialog box.

Examples

Add custom target architecture

Add custom target architecture defined in the XML file custom_arch.xml to the concurrent execution target architecture selector. This example requires you to create a custom_arch.xml first.

slexMulticoreExample
Simulink.architecture.register('custom_arch.xml')

Input Arguments

CustomArchFile — Custom target architecture file
XML file

Custom target architecture file that describes a custom target for concurrent execution, specified as an XML file.
See Also
Simulink.architecture.add | Simulink.architecture.delete |
Simulink.architecture.find_system |
Simulink.architecture.importAndSelect | Simulink.architecture.profile |
Simulink.architecture.set_param

Introduced in R2014a
Simulink.architecture.set_param

Set architecture object properties

Syntax

Simulink.architecture.set_param(Object,ParamName,ParamValue)

Description

Simulink.architecture.set_param(Object,ParamName,ParamValue) sets the specified parameter of Object to the specified value. Parameter name and value character vectors are case sensitive.

Examples

Set software node name

Set the software node name from MulticoreProcessor to MyCPUNewName.

slexMulticoreExample
Simulink.architecture.set_param([bdroot '/MulticoreProcessor'], 'Name', 'MyCPUNewName');

Change Periodic

Set Core2 trigger period to .01.

slexMulticoreExample
Simulink.architecture.set_param([bdroot '/MyCPUNewName/Core2'], 'Period', '.01')

Input Arguments

Object — Object whose parameter value to set
character vector
Object whose parameter value to set, specified as a character vector giving the object full path name. Possible objects are:

- Software node
- Hardware node
- Periodic trigger
- Aperiodic trigger
- Task

**ParamName** — **Name of the parameter to set**

character vector

Name of parameter whose value to set.

These are the possible parameters whose values you can set for each of the object types:

For software node:

- 'Name' — Name of the software node.

For hardware node:

- 'Name' — Name of the hardware node.
- 'ClockFrequency' — Frequency of the hardware node clock.
- 'Color' — Color of the trigger icon, specified as an RGB triplet (vector).

For a periodic trigger:

- 'Name' — Name of the trigger.
- 'Period' — Period of the trigger.
- 'Color' — Color of the trigger icon, specified as an RGB triplet (vector).

For an aperiodic trigger:

- 'Name' — Name of the trigger.
- 'Color' — Color of the trigger icon, specified as an RGB triplet (vector).
- 'EventHandlerType' — Trigger source for the interrupt-driven task. Possible values:
• 'Event (Windows)'
• 'Posix Signal (Linux/VxWorks 6.x)'
• 'SignalNumber' — Signal number for the trigger. You can set this value only if EventHandlerType is set to Event (Windows).
• 'EventName' — Event name for the trigger. You can set this value only if EventHandlerType is set to Posix Signal (Linux/VxWorks 6.x).

For task:
• 'Name' — Name of the task.
• 'Period' — Period of the task.
• 'Color' — Color of the task icon, specified as an RGB triplet (vector).

Data Types: char

ParamValue — Value to set the parameter to
calendar vector | vector

Value to set the parameter to, specified as a character vector, scalar, or vector. The possible values depend on the parameter.

Example: 'MyCPUNewName'

See Also
Simulink.architecture.add | Simulink.architecture.delete |
Simulink.architecture.find_system | Simulink.architecture.get_param |
Simulink.architecture.importAndSelect | Simulink.architecture.profile |
Simulink.architecture.register

Introduced in R2014a
Simulink.Block.getSampleTimes

Return sample time information for a block

Syntax

\[ ts = \text{Simulink.Block.getSampleTimes}(\text{block}) \]

Input Arguments

\textit{block}

Full name or handle of a Simulink block

Output Arguments

\textit{ts}

The command returns \textit{ts} which is a 1\(\times\)n array of Simulink.SampleTime objects associated with the model passed to Simulink.Block.getSampleTimes. Here \(n\) is the number of sample times associated with the block. The format of the returns is:

\[
1\times n \quad \text{Simulink.SampleTime} \\
\text{Package: Simulink} \\
\text{value: [1x2 double]} \\
\text{Description: [char string]} \\
\text{ColorRGBValue: [1x3 double]} \\
\text{Annotation: [char string]} \\
\text{OwnerBlock: [char string]} \\
\text{ComponentSampleTimes: [1x2 struct]} \\
\text{Methods}
\]

- \textit{value} — A two-element array of doubles that contains the sample time period and offset
- \textit{Description} — A character vector or string that describes the sample time type
- \textit{ColorRGBValue} — A 1x3 array of doubles that contains the red, green and blue (RGB) values of the sample time color
• **Annotation** — A character vector or string that represents the annotation of a specific sample time (e.g., ‘D1’)

• **OwnerBlock** — For asynchronous and variable sample times, a character vector or string containing the full path to the block that controls the sample time. For all other types of sample times, an empty character vector or string.

• **ComponentSampleTimes** — A structure array of elements of the same type as Simulink.BlockDiagram.getSampleTimes if the sample time is an async union or if the sample time is hybrid and the component sample times are available.

**Description**

```
ts = Simulink.Block.getSampleTimes(block) performs an update diagram and then returns the sample times of the block connected to the input argument mdl/signal. This method performs an update diagram to ensure that the sample time information returned is up-to-date. If the model is already in the compiled state via a call to the model API, then an update diagram is not necessary.
```

Using this method allows you to access all information in the Sample Time Legend programmatically.

**See Also**

Simulink.BlockDiagram.getSampleTimes

**Introduced in R2009a**
Simulink.BlockDiagram.addBusToVector

Convert virtual bus signals into vector signals by adding Bus to Vector blocks

Syntax

>[destBlocks,busToVectorBlocks,ignoredBlocks] = Simulink.BlockDiagram.addBusToVector(model)
>[destBlocks,busToVectorBlocks,ignoredBlocks] = Simulink.BlockDiagram.addBusToVector(model,includeLibs)
>[destBlocks,busToVectorBlocks,ignoredBlocks] = Simulink.BlockDiagram.addBusToVector(model,includeLibs,reportOnly)
>[destBlocks,busToVectorBlocks,ignoredBlocks] = Simulink.BlockDiagram.addBusToVector(model,includeLibs,reportOnly,strictOnly)

Description

>[destBlocks,busToVectorBlocks,ignoredBlocks] = Simulink.BlockDiagram.addBusToVector(model) searches a model, excluding any library blocks, for bus signals used implicitly as vectors, and returns the results of the search.

>[destBlocks,busToVectorBlocks,ignoredBlocks] = Simulink.BlockDiagram.addBusToVector(model,includeLibs) searches a model, and if includeLibs is true, includes in the search library blocks for bus signals used implicitly as vectors.

>[destBlocks,busToVectorBlocks,ignoredBlocks] = Simulink.BlockDiagram.addBusToVector(model,includeLibs,reportOnly) searches a model, and if reportOnly is set to false, then the function inserts a Bus to Vector block into each bus that is used as a vector in any block that it searches. The insertion replaces the implicit use of a bus as a vector with an explicit conversion of the bus to a vector. The source and destination blocks of the signal do not change.

If Simulink.BlockDiagram.addBusToVector adds Bus to Vector blocks to the model or any library, the function changes the saved copy of the diagram.
If Simulink.BlockDiagram.addBusToVector changes a library block, the change affects every instance of that block in every model that uses the library.

\[
[\text{destBlocks}, \text{busToVectorBlocks}, \text{ignoredBlocks}] = \text{Simulink.BlockDiagram.addBusToVector(model,includeLibs,reportOnly,strictOnly)}
\]

searches a model, and if strictOnly is true, the function checks for input bus signals used implicitly as vectors that are fed into one of these blocks. These blocks cannot take virtual bus signals, but they can accept nonvirtual bus signals.

- Delay
- Selector
- Assignment
- Vector Concatenate
- Reshape
- Permute Dimensions

**Examples**

**Identify Bus-To-Vector Conversions**

Model `ex_bus_to_vector` simulates correctly, but the input to the Gain block is a bus, while the output is a vector. The Gain block implicitly converts the bus to a vector.

Open the model.

\[
\text{open_system(fullfile(matlabroot,'examples','simulink','ex_bus_to_vector'))}
\]

Identify buses treated as vectors.
[blocks] = Simulink.BlockDiagram.addBusToVector(...
'ex_bus_to_vector')

### Processing block diagram 'ex_bus_to_vector'
### Number of blocks left that are connected to a bus being used as a vector: 2
### Done processing block diagram 'ex_bus_to_vector'

blocks =
1×2 struct array with fields:
    BlockPath
    InputPort
    LibPath

To understand the relationship between Simulink.BlockDiagram.addBusToVector and the **Bus signal treated as vector** configuration parameter, see “Manage Bus-to-Vector Conversions” on page 14-10.

**Add Bus to Vector Blocks**

Model `ex_bus_to_vector` simulates correctly, but the input to the Gain block is a bus, while the output is a vector. The Gain block implicitly converts the bus to a vector.

Open the model.

```matlab
open_system(fullfile(matlabroot,'examples','simulink','ex_bus_to_vector'))
```

![Block diagram of `ex_bus_to_vector` model](image)

Insert Bus to Vector blocks.

When you use function `Simulink.BlockDiagram.addBusToVector` with `reportOnly` set to `false`, the function saves the model. To create a writable copy of model `ex_bus_to_vector`, this example uses the `save_system` function.
save_system('ex_bus_to_vector','ex_bus_to_vector_blocks');
[blocks,busToVectors] = Simulink.BlockDiagram.addBusToVector(...
'ex_bus_to_vector_blocks',true,false);

The Gain block no longer implicitly converts the bus to a vector. The inserted Bus to Vector block performs the conversion explicitly. The Bus to Vector block is virtual and does not affect simulation results, code generation, or performance.

To understand the relationship between Simulink.BlockDiagram.addBusToVector and the Bus signal treated as vector configuration parameter, see “Manage Bus-to-Vector Conversions” on page 14-10.

**Input Arguments**

**model** — Model name or handle
character vector

Model name or handle, specified as a character vector.

**includeLibs** — Search library blocks
false (default) | true

Search library blocks, specified as false or true.

- **false** — Search only the blocks in the model.
- **true** — Search library blocks for bus signals used implicitly as vectors.

Specify as the second argument.

Data Types: logical

**reportOnly** — Report results without changing model
true (default) | false


Choice to report results without changing the model, specified as false or true.

- **false** — Update the model by inserting Bus to Vector blocks for bus signals that are implicitly used as vectors.
- **true** — Report search results, but do not change the model.

Specify as the third argument. Also specify the model and includeLibs arguments.

**Data Types:** logical

**strictOnly** — Check input bus signals used implicitly as vectors that feed blocks that can accept nonvirtual, but not virtual, bus signals

false (default) | true

Check input bus signals used implicitly as vectors that feed blocks that can accept nonvirtual, but not virtual, bus signals, specified as false or true. If **strictOnly** is true, the function checks for input bus signals used implicitly as vectors that are fed into one of these blocks. These blocks cannot take virtual bus signals, but they can accept nonvirtual bus signals.

- Delay
- Selector
- Assignment
- Vector Concatenate
- Reshape
- Permute Dimensions

Specify as the fourth argument. You must also specify the model, includeLibs, and reportOnly arguments.

**Data Types:** logical

**Output Arguments**

**destBlocks** — Blocks connected to buses but that treat buses as vectors

array of structures

Blocks connected to buses that treat buses as vectors, returned as an array of structures. Each structure in the array contains these fields:
• **BlockPath** — Character vector specifying the path to the block to which the bus connects.
• **InputPort** — Integer specifying the input port to which the bus connects.
• **LibPath** — If the block is a library block instance, and includeLibs is true, the field value is the path to the source library block. Otherwise, LibPath is empty ([]).

**busToVectorBlocks** — Bus to Vector blocks added by function

**cell array**

Bus to Vector blocks added by the function, specified as a cell array. If reportOnly is set to false, the cell array contains the paths to each Bus to Vector block that the function added to replace buses used as vectors. Otherwise, busToVectorBlocks is empty ([]).

**ignoredBlocks** — Cases where function cannot insert Bus to Vector block

**array of structures**

Cases where function cannot insert Bus to Vector block, specified as an array of structures. Each structure in the array contains these fields:

• **BlockPath** — Character vector specifying the path to the block to which the bus connects.
• **InputPort** — Integer specifying the input port to which the bus connects.

These cases occur when a Bus to Vector cannot be inserted because the input virtual bus signal consists of elements with mixed attributes.

**Tips**

• Before you execute this function:
  1. Ensure that the model compiles without error.
  2. Save the model.
• Back up the model and any libraries before calling the function with reportOnly set to false.
• To preview the effects of the change on blocks in all models, call Simulink.BlockDiagram.addBusToVector with includeLibs set to true and reportOnly set to true. Then, examine the information returned in the destBlocks output argument.
See Also
Bus to Vector

Introduced in R2007a
Simulink.BlockDiagram.arrangeSystem

Improve layout of model or subsystem

Syntax

Simulink.BlockDiagram.arrangeSystem(sys)

Description

Simulink.BlockDiagram.arrangeSystem(sys) improves model or subsystem layout by realigning, resizing, and moving blocks and straightening signal lines.

Examples

Arrange a Programmatically Populated Model

You can use Simulink.BlockDiagram.arrangeSystem with any open model, and it is particularly useful with models you populate programmatically.

Blocks in the model ex_arrange_system are poorly arranged.

open_system('ex_arrange_system')
Similarly, blocks in its subsystem are poorly arranged.

```matlab
open_system('ex_arrange_system/Subsystem')
```

Programmatically arrange blocks in the top layer of the specified systems.

```matlab
Simulink.BlockDiagram.arrangeSystem('ex_arrange_system')
```
Simulink.BlockDiagram.arrangeSystem('ex_arrange_system/Subsystem')

### Input Arguments

**sys** — Model name or subsystem block path

character vector

Model name or subsystem block path, specified as a character vector.

### See Also

Simulink.BlockDiagram.routeLine | add_block | add_line

### Topics

“Programmatic Modeling Basics”
“Adjust Visual Presentation to Improve Model Readability”

**Introduced in R2018a**
**Simulink.BlockDiagram.buildRapidAcceleratorTarget**

Build Rapid Accelerator target for model and return run-time parameter set

**Syntax**

```matlab
rtp = Simulink.BlockDiagram.buildRapidAcceleratorTarget(mdl)
```

**Description**

```matlab
rtp = Simulink.BlockDiagram.buildRapidAcceleratorTarget(mdl) builds a Rapid Accelerator target for model, mdl, and returns run-time parameter set, rtp.
```

**Input Arguments**

`mdl`

Name or handle of a Simulink model

**Output Arguments**

`rtp`

Run-time parameter set that contains two elements:

<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>modelChecksum</td>
<td>1x4 vector that encodes the structure of the model.</td>
</tr>
<tr>
<td>parameters</td>
<td>A structure of the tunable parameters in the model. This structure contains the following fields.</td>
</tr>
<tr>
<td>Field</td>
<td>Description</td>
</tr>
</tbody>
</table>

2-588
<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dataTypeName</td>
<td>The data type name, for example, double.</td>
</tr>
<tr>
<td>dataTypeId</td>
<td>Internal data type identifier for use by Simulink Coder.</td>
</tr>
<tr>
<td>complex</td>
<td>Complex type or real type specification. Value is 0 if real, 1 if complex.</td>
</tr>
<tr>
<td>dtTransIdx</td>
<td>Internal data type identifier for use by Simulink Coder.</td>
</tr>
<tr>
<td>values</td>
<td>All values associated with this entry in the parameters substructure.</td>
</tr>
<tr>
<td>map</td>
<td>Mapping structure information that correlates the values to the model tunable parameters. This structure contains the following fields.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Identifier</td>
<td>Tunable parameter name.</td>
</tr>
<tr>
<td>ValueIndices</td>
<td>Start and end indices into the values field, [startIdx, endIdx].</td>
</tr>
<tr>
<td>Dimensions</td>
<td>Dimension of this tunable parameter (matrices are generally stored in column-major format).</td>
</tr>
</tbody>
</table>

### Examples

**Build Rapid Accelerator Target for Model**

In the MATLAB Command Window, type:

```matlab
rtp = Simulink.BlockDiagram.buildRapidAcceleratorTarget('f14')
```

```matlab
### Building the rapid accelerator target for model: f14
### Successfully built the rapid accelerator target for model: f14
rtp =
```
See Also

Topics
"How Acceleration Modes Work"
"Choosing a Simulation Mode"
"Design Your Model for Effective Acceleration"

Introduced in R2012b
Simulink.BlockDiagram.copyContentsToSubsystem

Copy contents of block diagram to empty subsystem

Syntax

Simulink.BlockDiagram.copyContentsToSubsystem(bdiag, subsys)

Description

Simulink.BlockDiagram.copyContentsToSubsystem(bdiag, subsys) copies the contents of the block diagram bdiag to the subsystem subsys. The block diagram and subsystem must have already been loaded. The subsystem cannot be part of the block diagram.

The function affects only blocks, lines, and annotations; it does not affect nongraphical information such as configuration sets. You can use this function to convert a referenced model derived from an atomic subsystem into an atomic subsystem that is equivalent to the original subsystem.

This function cannot be used if the destination subsystem contains any blocks or signals. Other types of information can exist in the destination subsystem and are not affected by the function. Use Simulink.SubSystem.deleteContents if necessary to empty the subsystem before using Simulink.BlockDiagram.copyContentsToSubsystem.

Input Arguments

bdia
Block diagram name or handle

subsys
Subsystem name or handle
Examples

Copy the contents of vdp to an empty subsystem named vdp_subsystem that is in the model named new_model_with_vdp:

```matlab
open_system('vdp');
new_system('new_model_with_vdp')
open_system('new_model_with_vdp');
add_block('built-in/Subsystem', 'new_model_with_vdp/vdp_subsystem')
Simulink.BlockDiagram.copyContentsToSubsystem...
('vdp', 'new_model_with_vdp/vdp_subsystem')
```

See Also


Topics

“Non-Virtual Blocks” on page 13-10
“Create a Subsystem”

Introduced in R2007a
Simulink.BlockDiagram.createSubsystem

Create subsystem containing specified set of blocks

Syntax

Simulink.BlockDiagram.createSubsystem(blocks)
Simulink.BlockDiagram.createSubsystem()

Description

Simulink.BlockDiagram.createSubsystem(blocks) creates a new subsystem and moves the specified blocks into the subsystem. All of the specified blocks must originally reside at the top level of the model or in the same existing subsystem within the model.

If any of the blocks have unconnected input ports, the command creates input port blocks for each unconnected input port in the subsystem and connects the input port block to the unconnected input port. The command similarly creates and connects output port blocks for unconnected output ports on the specified blocks. If any of the specified blocks is an input port, the command creates an input port block in the parent system and connects it to the corresponding input port of the newly created subsystem. The command similarly creates and connects output port blocks for each of the specified blocks that is an output port block.

Simulink.BlockDiagram.createSubsystem() creates a new subsystem in the currently selected model and moves the currently selected blocks in the current model to the new subsystem.

Input Arguments

blocks

An array of block handles

Default: []
Examples

This function converts the contents of a model or subsystem into a subsystem.

```matlab
function convert2subsys(sys)
    blocks = find_system(sys, 'SearchDepth', 1);
    bh = [];
    for i = 2:length(blocks)
        bh = [bh get_param(blocks{i}, 'handle')];
    end
    Simulink.BlockDiagram.createSubsystem(bh);
end
```

For example, suppose you create this model and save it as `initial_model.slx`.

Execute this code to convert the model to create a subsystem:

```matlab
convert2subsys('initial_model');
```

See Also

Simulink.BlockDiagram.copyContentsToSubsystem
Simulink.BlockDiagram.deleteContents
Simulink.SubSystem.convertToModelReference
Simulink.SubSystem.copyContentsToBlockDiagram

Topics
“Non-Virtual Blocks” on page 13-10
“Create a Subsystem”

Introduced in R2009a
Simulink.BlockDiagram.deleteContents

Delete contents of block diagram

Syntax

Simulink.BlockDiagram.deleteContents(bdiag)

Description

Simulink.BlockDiagram.deleteContents(bdiag) deletes the contents of the block diagram bdiag. The function affects only blocks, lines, and annotations. The block diagram must have already been loaded.

Input Arguments

bdiag

Block diagram name or handle

Examples

Delete the graphical content of an open block diagram named f14, including all subsystems:

load_system('f14')
Simulink.BlockDiagram.deleteContents('f14');

See Also

Topics

“Modeling”
“Create a Subsystem”

Introduced in R2007a
Simulink.BlockDiagram.getAlgebraicLoops

Show algebraic loops in a model

Syntax

Simulink.blockDiagram.getAlgebraicLoops(model)
[loops,h] = Simulink.BlockDiagram.getAlgebraicLoops(model)

Description

Simulink.blockDiagram.getAlgebraicLoops(model) returns a list of algebraic loops in a model as a vector of AlgebraicLoops objects.

The Algebraic Loops viewer provides a tabular list of all algebraic loops in the model. Use the Algebraic Loops viewer to control the display of algebraic loops in your model.

[loops,h] = Simulink.BlockDiagram.getAlgebraicLoops(model)

Examples

Highlight all algebraic loops in model

Open the ex_sldemo_hydcyl model.

In the Diagnostics pane of Model Configuration Parameters, set Algebraic loop to none or warning. Setting this parameter to error prevents the model from compiling.

Compile the model without any errors. The model must compile before you can highlight any algebraic loops.

At the MATLAB command prompt, enter:

mdl = 'ex_sldemo_hydcyl';
load_system(mdl);
Simulink.BlockDiagram.getAlgebraicLoops(bdroot)
The resulting output shows that the model has one algebraic loop, which contains three blocks.

Running the `getAlgebraicLoops` function also opens the Algebraic Loop viewer. Use the Algebraic loop viewer to highlight specific algebraic loops in your model or all the loops.

The selected loops are highlighted in the corresponding model, in.
### Input Arguments

**model** — Model to highlight  
string scalar | character vector

Name of the model being tested, specified as a string scalar or character vector. You can also pass the most recently opened model using `bdroot`.

**Note** The model must be loaded into memory using either `open_system` or `load_system`.

Data Types: char | string

### Output Arguments

**loops** — Algebraic loops  
vector of `AlgebraicLoop` objects

Every algebraic loop in the model returned as an `AlgebraicLoop` object that contains the following information:
• **Model**—`Simulink.BlockDiagram` object representing the model
• **Id**—Index of the algebraic loop in the model
• **VariableBlockHandles**—Handles to blocks that have been assigned algebraic variables
• **BlockHandles**—Handles to all the blocks contained in the algebraic loop
• **IsArtificial**—Boolean value indicating whether the loop is an artificial algebraic loop.

**h** — **Handle to control UI**
`matlab.ui.Figure` handle

The control UI is a MATLAB figure window. Use the UI's handle h to control it programmatically. For more information, see Figure Properties.

**See Also**
ashow

**Topics**
“Identify Algebraic Loops in Your Model”
“Remove Algebraic Loops”

**Introduced before R2006a**
Simulink.BlockDiagram.expandSubsystem

Replace subsystem with subsystem contents

**Syntax**

Simulink.BlockDiagram.expandSubsystem(block)

Simulink.BlockDiagram.expandSubsystem(block,'CreateArea',specifier)

**Description**

Simulink.BlockDiagram.expandSubsystem(block) expands the subsystem block into the system that contains the subsystem. Open or load the model first.

You can expand virtual subsystems that are not masked, linked, or commented. For details, see “Subsystems That You Can Expand”.

Simulink.BlockDiagram.expandSubsystem(block,'CreateArea',specifier) prevents creating an area around the expanded subsystem.

**Input Arguments**

**block** — Subsystem to expand

block path name | block handle

Subsystem to expand, specified as the block path name or block handle.

Example: 'sldemo_enginewc/Combustion'

**specifier** — Specify whether to create an area

'On' (default) | 'Off'

Option to specify whether to create an area, specified as 'On' to create the area, or 'Off' to not create the area.
Examples

Expand Subsystem

Open the model sldemo_enginewc. Expand the Combustion subsystem.

```matlab
open_system('sldemo_enginewc')
Simulink.BlockDiagram.expandSubsystem('sldemo_enginewc/Combustion')
```

The subsystem contents appear in the top level of the model.

Expand Subsystem Without Creating an Area

Expand the Combustion subsystem. The blocks and signals in the Combustion subsystem become part of the system and replace the Subsystem block.

Open the model sldemo_enginewc. Expand the Combustion subsystem but do not create an area.

```matlab
open_system('sldemo_enginewc')
Simulink.BlockDiagram.expandSubsystem('sldemo_enginewc/Combustion','CreateArea','Off')
```

See Also


Topics

“Expand Subsystem Contents”
“Non-Virtual Blocks” on page 13-10

Introduced in R2014a
Simulink.BlockDiagram.getChecksum

Return checksum of model

Syntax

\[
[\text{checksum}, \text{details}] = \text{Simulink.BlockDiagram.getChecksum('model')}\]

Description

\[
[\text{checksum}, \text{details}] = \text{Simulink.BlockDiagram.getChecksum('model')}\]
returns the checksum of the specified model. Simulink software computes the checksum based on attributes of the model and the blocks the model contains.

One use of this command is to determine why the Accelerator mode in Simulink software regenerates code. For an example, see slAccelDemoWhyRebuild.

**Note** Simulink.BlockDiagram.getChecksum compiles the specified model, if the model is not already in a compiled state.

This command accepts the argument *model*, which is the full name or handle of the model for which you are returning checksum data.

This command returns the following output:

- **checksum** — Array of four 32-bit integers that represents the model's 128-bit checksum.
- **details** — Structure of the form

  ContentsChecksum: [1x1 struct]
  InterfaceChecksum: [1x1 struct]
  ContentsChecksumItems: [nx1 struct]
  InterfaceChecksumItems: [mx1 struct]

  - ContentsChecksum — Structure of the following form that represents a checksum that provides information about all blocks in the model.
Value: [4x1 uint32]
MarkedUnique: [bool]

- **Value** — Array of four 32-bit integers that represents the model's 128-bit checksum.
- **MarkedUnique** — True if any blocks in the model have a property that prevents code reuse.

**InterfaceChecksum** — Structure of the following form that represents a checksum that provides information about the model.

Value: [4x1 uint32]
MarkedUnique: [bool]

- **Value** — Array of four 32-bit integers that represents the model's 128-bit checksum.
- **MarkedUnique** — Always true. Present for consistency with **ContentsChecksum** structure.

**ContentsChecksumItems** and **InterfaceChecksumItems** — Structure arrays of the following form that contain information that Simulink software uses to compute the checksum for **ContentsChecksum** and **InterfaceChecksum**, respectively:

Handle: [char array]
Identifier: [char array]
Value: [type]

- **Handle** — Object for which Simulink software added an item to the checksum. For a block, the handle is a full block path. For a block port, the handle is the full block path and a character vector that identifies the port.
- **Identifier** — Descriptor of the item Simulink software added to the checksum. If the item is a documented parameter, the identifier is the parameter name.
- **Value** — Value of the item Simulink software added to the checksum. If the item is a parameter, **Value** is the value returned by `get_param(handle, identifier)`

**Simulink.BlockDiagram.getChecksum** returns a checksum that depends on why and how you compiled the model. This function also compiles the model if it is not in a compiled state. The model compiles for:
• Simulation— if the simulation mode is Accelerator or you have not installed Simulink Coder
• Code generation— in all other cases

To compile the model before calling Simulink.BlockDiagram.getChecksum, use this command:

`modelName([],[],[],'compile')`

**Note** The checksum that Simulink.BlockDiagram.getChecksum returns can vary from the checksum returned if you first compile the model at the command line (using the `model` command) before running Simulink.BlockDiagram.getChecksum.

**Tip**

The structural checksum reflects changes to the model that can affect the simulation results, including:

• Changing the solver type, for example from Variable-step to Fixed-step
• Adding or deleting blocks or connections between blocks
• Changing the values of nontunable block parameters, for example, the **Seed** parameter of the Random Number block
• Changing the number of inputs or outputs of blocks, even if the connectivity is vectorized
• Changing the number of states or the initial states in the model
• Selecting a different function in the Trigonometric Function block
• Changing signs used in a Sum block
• Adding a Target Language Compiler (TLC) file to inline an S-function

Examples of model changes that do not affect the structural checksum include:

• Changing the position of a block
• Changing the position of a line
• Resizing a block
• Adding, removing, or changing a model annotation
See Also
Simulink.SubSystem.getChecksum | Simulink.getFileChecksum

Introduced in R2006b
Simulink.BlockDiagram.getInitialState

Return initial state data of block diagram

Syntax

x0 = Simulink.BlockDiagram.getInitialState('model')

Description

x0 = Simulink.BlockDiagram.getInitialState('model') returns the initial state data of the block diagram specified by the input argument model. You can use this initial state data as the initial state for simulating a model or to provide an initial state condition to the linearization commands. To specify the initial state for a simulation, use the LoadInitialState model argument or the Data Import/Export > Initial state configuration parameter.

To specify the format for the initial state data, use the SaveFormat model argument. The default format is 'Dataset'. Other formats 'Array', 'Structure', and 'StructureWithTime'. Alternatively, you can set the initial state format using the Data Import/Export > Format configuration parameter.

- If format is 'Dataset', then the Simulink.BlockDiagram.getInitialState function returns a Simulink.SimulationData.Dataset object.
- For other format settings, the function returns a structure of the form:

  time: 0
  signals: [1xn struct]

  where n is the number of states contained in the model, including any models referenced by Model blocks. The signals field is a structure of the form:

  values: [1xm double]
  dimensions: [1x1 double]
  label: [char array]
  blockName: [char array]
  inReferencedModel: [bool]
  sampleTime: [1x2 double]
• **values** — Numeric array of length m, where m is the number of states in the signal
• **dimensions** — Length of the values vector
• **label** — Indication of whether the state is continuous (CSTATE) or discrete. If the state is discrete:

  The name of the discrete state is shown for S-function blocks.

  The name of the discrete state is shown for those built-in blocks that assign their own names to discrete states.

  DSTATE is used in all other cases.

• **blockName** — Full path to block associated with this state
• **inReferencedModel** — Indication of whether the state originates in a model referenced by a Model block (1) or in the top model (0)
• **sampleTime** — Array containing the sample time and offset of the block that owns the state

Using this function to return the initial state data simplifies specifying initial state values for models with multiple states. Each state is associated with the full path to its parent block.

**See Also**
linmod

**Topics**
“Initial state”
“Format”

**Introduced in R2006b**
Simulink.BlockDiagramSAMPLE_TIMES

Return all sample times associated with model

Syntax

ts = Simulink.BlockDiagram.getSampleTimes('model')

Description

ts = Simulink.BlockDiagram.getSampleTimes('model') performs an update diagram and then returns the sample times associated with the block diagram model. The update diagram ensures that the sample time information returned is up to date. If the model is already in the compiled state via a call to the model API, then an update diagram is not necessary.

Using this method allows you to access all information in the Sample Time Legend programmatically.

Input Arguments

model
    Name or handle of a Simulink model

Output Arguments

ts
    The command returns a 1xn array of Simulink.SampleTime objects associated with the model passed to Simulink.BlockDiagram.getSampleTimes. Here n is the number of sample times associated with the block diagram. The format of the returns is as follows:

    1xn Simulink.SampleTime

Package: Simulink
value: [1x2 double]
Description: [char string]
ColorRGBValue: [1x3 double]
Annotation: [char string]
OwnerBlock: [char string]
ComponentSampleTimes: [1x2 struct]

Methods

• value — A two-element array of doubles that contains the sample time period and offset.
• Description — A character vector or string that describes the sample time type.
• ColorRGBValue — A 1x3 array of doubles that contains the red, green, and blue (RGB) values of the sample time color.
• Annotation — A character vector or string that represents the annotation of a specific sample time (e.g., 'D1').
• OwnerBlock — For asynchronous and variable sample times, a character vector or string containing the full path to the block that controls the sample time. For all other types of sample times, an empty character vector or string.
• ComponentSampleTimes — A structure array of elements of the same type as Simulink.BlockDiagram.getSampleTimes if the sample time is an async union or if the sample time is hybrid and the component sample times are available.

See Also
Simulink.Block.getSampleTimes

Introduced in R2009a
Simulink.BlockDiagram.loadActiveConfigSet

**Package:** Simulink.BlockDiagram

Load, associate, and activate configuration set with model

**Syntax**

Simulink.BlockDiagram.loadActiveConfigSet(model, filename)

**Description**

Simulink.BlockDiagram.loadActiveConfigSet(model, filename) loads a configuration set, associates it with a model, and makes it the active configuration set. model is the name or handle of a model. filename is the name of the file (.m or .mat) that creates or contains a configuration set object to load. If you do not provide a file extension, it defaults to .m. If the file name is the same as a model name on the MATLAB path, the software cannot determine which file contains the configuration set object and displays an error message.

**Examples**

Save the configuration set from the sldemo_counters model to my_config_set.m.

% Open the sldemo_counters model  
sldemo_counters  
% Save the active configuration set to my_config_set.m  
Simulink.BlockDiagram.saveActiveConfigSet('sldemo_counters', 'my_config_set.m')

Load the configuration set from my_config_set.m, associate it with the vdp model, and make it the active configuration set.

% Open the vdp model  
vdp  
% Load the configuration set from my_config_set.m, making it the active  
% configuration set for vdp.  
Simulink.BlockDiagram.loadActiveConfigSet('vdp', 'my_config_set.m')
Tips

• If you load a configuration set with the same name as the active configuration set, the software overwrites the active configuration set.

• If you load a configuration set with the same name as an inactive configuration set associated with the model, the software detaches the inactive configuration from the model.

• If you load a configuration set object that contains an invalid custom target, the software sets the “System target file” (Simulink Coder) parameter to ert.tlc.

See Also
Simulink.BlockDiagram.saveActiveConfigSet | Simulink.ConfigSet | attachConfigSet | attachConfigSetCopy | detachConfigSet | getActiveConfigSet | getConfigSet | getConfigSets | setActiveConfigSet

Topics
“Load a Saved Configuration Set”

Introduced in R2010b
Simulink.BlockDiagram.propagateConfigSet

Propagate top model configuration reference to referenced models

Syntax

```
[isPropagated, convertedModels] = Simulink.BlockDiagram.propagateConfigSet(model)
[isPropagated, convertedModels] = Simulink.BlockDiagram.propagateConfigSet(model, 'include', refModels)
[isPropagated, convertedModels] = Simulink.BlockDiagram.propagateConfigSet(model, 'exclude', refModels)
handle = Simulink.BlockDiagram.propagateConfigSet(model, 'gui')
```

Description

- `[isPropagated, convertedModels] = Simulink.BlockDiagram.propagateConfigSet(model)` propagates the configuration reference for `model` to all referenced models. Execute the function from a writable folder.

- `[isPropagated, convertedModels] = Simulink.BlockDiagram.propagateConfigSet(model, 'include', refModels)` propagates the configuration reference for `model` to the models in the `refModels` list. Execute the function from a writable folder.

- `[isPropagated, convertedModels] = Simulink.BlockDiagram.propagateConfigSet(model, 'exclude', refModels)` propagates the configuration reference for `model` to all referenced models in the hierarchy except for the models in the `refModels` list. Execute the function from a writable folder.

- `handle = Simulink.BlockDiagram.propagateConfigSet(model, 'gui')` opens the Configuration Reference Propagation to Referenced Models dialog box.
Examples

Propagate a Configuration Reference to All Referenced Models

```matlab
[isPropagated,convertedModels] = ... Simulink.BlockDiagram.propagateConfigSet('sldemo_mdlref_depgraph')
```

Propagate a Configuration Reference to Listed Referenced Models

```matlab
[isPropagated,convertedModels] = ... Simulink.BlockDiagram.propagateConfigSet(...
'sldemo_mdlref_depgraph','include',...
{'sldemo_mdlref_heater','sldemo_mdlref_house'})
```

Propagate a Configuration Reference to Referenced Models with Exclusions

```matlab
[isPropagated,convertedModels] = ... Simulink.BlockDiagram.propagateConfigSet(...
'sldemo_mdlref_depgraph','exclude',...
{'sldemo_mdlref_heater','sldemo_mdlref_house'})
```

Open the Configuration Reference Propagation to Referenced Models Dialog Box for a Model

```matlab
Simulink.BlockDiagram.propagateConfigSet(...
'sldemo_mdlref_depgraph','gui')
```

Input Arguments

`model` — Top model
character vector | string scalar

Top model with configuration reference to propagate, specified as a character vector or string scalar:
Example: `‘mdl’`
**refModels — Referenced models**
cell array of character vectors | string array

List of referenced models to be included or excluded in propagation, specified as a cell array of character vectors or string array.
Example: {'mdl1','mdl2','mdl3'}

**Output Arguments**

**isPropagated — Success of propagation**
false (default) | true

Indication of whether configuration reference propagation is successful, specified as a Boolean.

**convertedModels — Converted models**
cell array of character vectors

List of converted model names, specified as a cell array of character vectors.

**handle — Handle to dialog box**
handle

Handle to the `Configuration Reference Propagation to Referenced Models` dialog box. Returned when you specify the ‘gui’ argument to the function.

**See Also**
Simulink.BlockDiagram.restoreConfigSet

**Topics**
“Share a Configuration Across Referenced Models”
“Manage a Configuration Reference”

**Introduced in R2012b**
Simulink.BlockDiagram.restoreConfigSet

Restore model configuration for converted models

Syntax

\[ [\text{isRestored}, \text{restoredModels}] = \text{Simulink.BlockDiagram.restoreConfigSet}(\text{model}) \]

Description

\[ [\text{isRestored}, \text{restoredModels}] = \text{Simulink.BlockDiagram.restoreConfigSet}(\text{model}) \]
restores the model configuration for all converted models after propagating a configuration reference from a top model to the referenced models. Execute the function from a writable folder.

Examples

Restore the Model Configuration for Converted Models

\[ [\text{isRestored}, \text{restoredModels}] = \ldots \]
\[ \text{Simulink.BlockDiagram.restoreConfigSet('sldemo_mdlref_depgraph')}; \]

Input Arguments

\textbf{model — Top model}  
character vector | string scalar

Name of top model, specified as a character vector or string scalar.

Example: ‘mdl’
Output Arguments

**isRestored — Success of restoration**
false (default) | true

Indication of whether configuration reference propagation is successful, specified as a Boolean.

**restoredModels — Restored models**

cell array of character vectors

List of restored model names, specified as a cell array of character vectors.

See Also

Simulink.BlockDiagram.propagateConfigSet

Topics

“Share a Configuration Across Referenced Models”
“Manage a Configuration Reference”

Introduced in R2012b
Simulink.BlockDiagram.routeLine

Route existing lines of a model

Syntax

Simulink.BlockDiagram.routeLine(lineHandles)
Simulink.BlockDiagram.routeLine('DeferredLines')

Description

Simulink.BlockDiagram.routeLine(lineHandles) takes an array of existing line handles as input and routes the corresponding lines. Routing existing lines improves line route quality and avoids overlaps of a line with other lines and obstacles in the model.

To find the line handles in the model you want to route, use the find_system or get_param function.

Simulink.BlockDiagram.routeLine('DeferredLines') performs all line updates that the LineUpdate model parameter defers while a script runs. For this command to route lines, all of these conditions must apply:

- The LineUpdate model parameter is set to 'deferred'.
- The script that contains this command affects line routing before using this command.
- If the script saves models, the script affects line routing after using the save_system function and before using this command.

Examples

Route Existing Lines of Model

This example shows how to route existing lines of a model that overlap with other lines and blocks in the model.
Open the model `ex_route_lines`.

```matlab
open_system('ex_route_lines');
```

The two lines in the top-level system overlap with each other and with a block. To route these lines, create an array of line handles by using `find_system`. Then call `Simulink.BlockDiagram.routeLine` with this array as input.

```matlab
lineHandles = find_system(gcs,'FindAll','On','SearchDepth',1,'Type','Line'); Simulink.BlockDiagram.routeLine(lineHandles);
```

The updated model has no overlaps.
Route Lines Affected by Script

When the LineUpdate model parameter is set to 'deferred', scripts postpone line routing until they pause, complete, or run the Simulink.BlockDiagram.routeLine('DeferredLines') command.

Model ex_route_deferred_lines contains virtual subsystems with numerous signal lines connecting them.

To see how the Simulink.BlockDiagram.routeLine('DeferredLines') command works with the LineUpdate model parameter, set this parameter to 'deferred'.

set_param(0,'LineUpdate','deferred');

Capture Image Before Routing Deferred Lines

Run a MATLAB® script that loads the model, moves blocks within the model, and saves an image of the model.

load_system('ex_route_deferred_lines');
set_param([gcs '/Subsystem2'],'Position',[765 -370 855 -330]);
set_param([gcs '/Subsystem3'],'Position',[300 215 390 335]);
set_param([gcs '/Subsystem4'],'Position',[270 -135 360 -95]);
print('-s','deferred_lines_unrouted','-dpng');
Because the script saves an image of the model before the deferred lines are routed, lines overlap other lines and obstacles in the model.

**Capture Image After Routing Deferred Lines**

Run a MATLAB script that loads the original model, moves blocks within the model, updates affected line routing, and saves an image of the model.

```matlab
load_system('ex_route_deferred_lines');
set_param([gcs '/Subsystem2'],'Position',[765 -370 855 -330]);
set_param([gcs '/Subsystem3'],'Position',[300 215 390 335]);
set_param([gcs '/Subsystem4'],'Position',[270 -135 360 -95]);
Simulink.BlockDiagram.routeLine('DeferredLines');
print('-s','deferred_lines_routed','-dpng');
```
Adding command `Simulink.BlockDiagram.routeline('DeferredLines')` before the `print` function improves line routing in the saved image.

**Input Arguments**

`lineHandles` — Handles of lines to route
array of line handles

Handles of the lines to route, specified as an array of line handles. To find the line handles in your model, use the `find_system` or `get_param` function.

Data Types: `double`

**Limitations**

- Line routing can require additional empty space on the Simulink canvas. If the canvas does not have the required empty space, using `Simulink.BlockDiagram.routeLine` might not improve line route quality.
• For large models with many lines to route, `Simulink.BlockDiagram.routeLine` can take several minutes to execute.

**See Also**

`Simulink.BlockDiagram.arrangeSystem` | `find_system` | `get_param`

**Topics**

“Programmatic Modeling Basics”
“Adjust Visual Presentation to Improve Model Readability”

**Introduced in R2019a**
Simulink.BlockDiagram.saveActiveConfigSet

**Package:** Simulink.BlockDiagram

Save active configuration set of model

**Syntax**

Simulink.BlockDiagram.saveActiveConfigSet(model, filename)

**Description**

Simulink.BlockDiagram.saveActiveConfigSet(model, filename) saves the active configuration set of a model to a .m or .mat file. **model** is the name or handle of the model. **filename** is the name of the file to save the model configuration set. If you specify a .m extension, the file contains a function that creates a configuration set object. If you specify a .mat extension, the file contains a configuration set object. If you do not provide a file extension, the active configuration set is saved to a file with a .m extension. Do not specify **filename** to be the same as a model name; otherwise, the software cannot determine which file contains the configuration set object when loading the file.

**Note** If you specify a .mat extension when you save the active configuration set, all of the parameters are preserved. If you specify a .m extension, the .m file does not include hidden or disabled parameters.

**Examples**

Save the configuration set from the sldemo_counters model to my_config_set.m.

```matlab
% Open the sldemo_counters model
sldemo_counters
% Save the active configuration set to my_config_set.m
Simulink.BlockDiagram.saveActiveConfigSet('sldemo_counters', 'my_config_set.m')
```
See Also
Simulink.BlockDiagram.loadActiveConfigSet | Simulink.ConfigSet | attachConfigSet | attachConfigSetCopy | detachConfigSet | getActiveConfigSet | getConfigSet | getConfigSets | setActiveConfigSet

Topics
“Save a Configuration Set”

Introduced in R2010b
Simulink.Bus.cellToObject

Package: Simulink.Bus

Convert cell array containing bus information to bus objects

Syntax

Simulink.Bus.cellToObject(busCells)

Description

Simulink.Bus.cellToObject(busCells) creates a set of bus objects in the MATLAB base workspace from a cell array of bus information.

Examples

Create Bus Objects from Cell Array of Bus Information

Create a cell array of cell arrays of bus information, and use that cell array to generate a bus object in the base workspace.

Create a cell array of bus information.

busCell = { ... 
    { ... 
        'myBusObj', ... 
        'MyHeader.h', ... 
        'My description', ... 
        'Exported', ... 
        '-1', ... 
       ('{{'a',1,'double', [0.2 0],'real','Sample'}; ... 
        {{'b',1,'double', [0.2 0],'real','Sample',... 
        'Fixed','-3,3','m','b is distance from the origin'}}),... 
    }, ... 
};
Generate a bus object in the base workspace from the cell array.

Simulink.Bus.cellToObject(busCell)

### Input Arguments

**busCells — Bus object information**

cell array of cell arrays

Bus object information, specified as a cell array of cell arrays. Each subordinate cell array must contain this bus object information:

1. Bus name
2. Header file
3. Description
4. Data scope
5. Alignment
6. Elements

The elements field is an array that must contain this data for each element:

1. Element name
2. Dimensions
3. Data type
4. Sample time
5. Complexity
6. Sampling mode

The elements field array can also contain this data:

1. Dimensions mode
2. Minimum
3. Maximum
4. Units
5. Description
Tips

The inverse function is `Simulink.Bus.objectToCell`.

Compatibility Considerations

**Simulink.BusElement objects will no longer support the SampleTime property**

_Not recommended starting in R2019b_

In a future release, the `SampleTime` property of `Simulink.BusElement` objects will be removed.

To prepare:

- When defining `BusElement` objects, use the default `SampleTime` value (-1).
- To specify the sample time for an element of a bus, use the `SampleTime` block parameter of corresponding blocks. For example, you can use In Bus Element, Out Bus Element, and Signal Specification blocks to specify sample time.

**Simulink.BusElement objects will no longer support the SamplingMode property**

_Not recommended starting in R2016b_

In R2016b, the `SamplingMode` property of `Simulink.BusElement` objects was removed. Scripts that use the `SamplingMode` property of `Simulink.BusElement` objects continue to work. `Simulink.Bus.cellToObject` continues to require the `SamplingMode` field and `Simulink.Bus.objectToCell` continues to include the sampling mode in the output cell arrays.

In a future release, support for the `SamplingMode` property will be removed.

To specify whether a signal is sample-based or frame-based, define the sampling mode of input signals at the block level instead of at the signal level.
See Also

Classes
Simulink.Bus | Simulink.BusElement

Functions

Topics
“Specify Bus Properties with Bus Objects”
“Create Bus Objects Programmatically”
“Create and Specify Bus Objects”
“Save Bus Objects”

Introduced before R2006a
Simulink.Bus.createMATLABStruct

Package: Simulink.Bus

Create MATLAB structures using same hierarchy and attributes as bus signals

Syntax

structFromBus = Simulink.Bus.createMATLABStruct(busSource)
structFromBus = Simulink.Bus.createMATLABStruct(busSource, partialValues)
structFromBus = Simulink.Bus.createMATLABStruct(busSource, partialValues, dims)

structsForBuses = Simulink.Bus.createMATLABStruct(portHandles)
structsForBuses = Simulink.Bus.createMATLABStruct(portHandles, partialStructures)
structsForBuses = Simulink.Bus.createMATLABStruct(busObjectNames)

Description

structFromBus = Simulink.Bus.createMATLABStruct(busSource) creates a MATLAB structure that has the same hierarchy and attributes (such as type and dimension) as the bus specified in busSource. The resulting structure uses the ground values of the bus signal.

structFromBus = Simulink.Bus.createMATLABStruct(busSource, partialValues) creates a structure that uses specified values of partialValues instead of the corresponding ground values of the bus signal.

structFromBus = Simulink.Bus.createMATLABStruct(busSource, partialValues, dims) creates a structure that has the specified dimensions. To create a structure for an array of buses, use dims.

structsForBuses = Simulink.Bus.createMATLABStruct(portHandles) creates a cell array of structures for bus signal ports, specified with port handles. The resulting cell array of structures uses ground values. Use this syntax to create initialization
structures for multiple bus ports. This syntax improves performance compared to using separate Simulink.Bus.createMATLABStruct calls to create the structures.

\[
\text{structsForBuses} = \text{Simulink.Bus.createMATLABStruct}(\text{portHandles}, \text{partialStructures})
\]
creates a cell array of structures that uses the specified values of \text{partialStructures} instead of the ground values.

\[
\text{structsForBuses} = \text{Simulink.Bus.createMATLABStruct}(\text{busObjectNames})
\]
creates a cell array of structures based on the specified bus objects.

**Examples**

**Create MATLAB Structure from Bus Object**

Open and simulate model \text{ex_bus_initial_conditions}.

\[
\text{open_system('ex_bus_initial_conditions')} \\
\text{sim('ex_bus_initial_conditions');}
\]
Create a MATLAB structure using bus object `Top`, which model
`ex_bus_initial_conditions` loads.

```matlab
mStruct = Simulink.Bus.createMATLABStruct('Top');
```

Set a value for the field of the `mStruct` structure that corresponds to bus element `A1` of
bus `A`.

```matlab
mStruct.A.A1 = 3;
```

```matlab
ans =

    struct with fields:
```

2-633
Simulink sets the other fields in the structure to the ground values of the corresponding bus elements.

You can use `mStruct` as the initial condition structure for the Unit Delay block.

### Initialize Signal Elements that Use a Data Type Other than Double

Create a MATLAB structure for a bus whose signal elements use a data type other than `double`. Use a partial structure to specify initialization values for a subset of the elements. When you create the partial structure, match the data types of the fields with the data types of the corresponding elements.

Open and simulate model `ex_bus_initial_conditions`.

```matlab
open_system('ex_bus_initial_conditions')
sim('ex_bus_initial_conditions');
```
The C1 signal element that the block labeled Constant5 produces uses the data type int16.

Find the port handle for the Bus Creator block port that produces the Top bus signal.

\[ \text{ph} = \text{get_param('ex_bus_initial_conditions/TopBus','PortHandles');} \]

Create a partial structure that specifies values for a subset of the elements in the bus signal created by the TopBus block. To set the value of the C.C1 field, use a typed expression. Match the data type in the expression with the data type of the signal element in the model (int16).

\[ \text{PartialstructForK} = \text{struct('B',3,'C',struct('C1',\text{int16}(5))}); \]

Create a full structure by using the port handle (ph) for the TopBus block. Override the ground values for the C.C1 and B elements.
outPort = ph.Outport;
mStruct = Simulink.Bus.createMATLABStruct(outPort,PartialstructForK);

The field C.C1 in the output structure continues to use the data type int16.

**Create MATLAB Structure with Specified Dimensions**

Open and simulate model ex_bus_initial_conditions.

```matlab
open_system('ex_bus_initial_conditions')
sim('ex_bus_initial_conditions');
```

Create a partial structure for a subset of bus elements in the bus created by the TopBus block.
PartialStructForK = struct('A',struct('A1',4),'B',3)

PartialStructForK =
    struct with fields:
    A: [1x1 struct]
    B: 3

Create a MATLAB structure using bus object Top, a partial structure, and dimensions for the resulting structure.

structFromBus = Simulink.Bus.createMATLABStruct...
               ('Top',PartialStructForK,[2 3])

structFromBus =
    2x3 struct array with fields:
    A
    B
    C

Create Cell Array of MATLAB Structures

To create initialization structures for multiple bus ports, specify port handles as arguments for Simulink.Bus.createMATLABStruct. The resulting cell array of structures uses ground values.

Open and simulate model ex_two_outports_create_struct.

open_system('ex_two_outports_create_struct')
sim('ex_two_outports_create_struct');
Find the port handles for the Bus Creator blocks Bus1 and Bus2.

```matlab
ph_1 = get_param...
     ('ex_two_outports_create_struct/Bus Creator','PortHandles');
ph_2 = get_param...
     ('ex_two_outports_create_struct/Bus Creator1','PortHandles');

Create a MATLAB® structure using an array of port handles.

mStruct = Simulink.Bus.createMATLABStruct([ph_1.Outport ph_2.Outport])

mStruct =

2x1 cell array

{1x1 struct}
{1x1 struct}
Create MATLAB Structure from Bus Port and Partial Structure

Create a MATLAB structure based on a port that connects to a bus signal. Use a partial structure to specify values for a subset of the bus elements in the bus that connects to the port.

Open and simulate model `ex_bus_initial_conditions`.

```matlab
open_system('ex_bus_initial_conditions')
sim('ex_bus_initial_conditions');
```

Find the port handle for the Bus Creator block port that produces the Top bus signal. The `Outport` handle is the handle that you need.

```matlab
ph = get_param('ex_bus_initial_conditions/TopBus','PortHandles')
```
Create a partial structure for the bus signal created by the TopBus block. You can use a partial structure to specify values for a subset of bus elements.

```
PartialStructForK = struct('A',struct('A1',4),'B',3)
```

Bus elements represented by structure fields Top.B and Top.A are at the same level in the bus hierarchy. You can use this partial structure to override the ground values for the B and A bus signal elements.

When you create a structure from a bus object or from a bus port, you can use a partial structure as an optional argument.

Create a MATLAB structure by using the port handle (ph) for the TopBus block. Override the ground values for the A.A1 and B bus elements.

```
outPort = ph.Outport;
mStruct = Simulink.Bus.createMATLABStruct(outPort,PartialStructForK)
```
struct with fields:

A: [1x1 struct]
B: 3
C: [1x1 struct]

**Input Arguments**

**busSource** — Source representing a bus signal

A Simulink.Bus object | port handle

Source representing a bus signal to use for creating a MATLAB structure, specified as the name of a bus object or port handle.

- If you use the `dims` argument, then for `busSource`, use a bus object.
- For an array of buses signal, you cannot use a port handle.
- If you use a bus object name, then the bus object must be in the MATLAB base workspace. The data type for the bus object name is `char`.
- If you use a port handle, then the model must compile successfully before you use the `Simulink.Bus.createMATLABStruct` method. The data type for the port handle is a `double`.

Example: `structFromBus = Simulink.Bus.createMATLABStruct('myBusObject')`

Example: `structForPortHandle = Simulink.Bus.createMATLABStruct(port_handle_1)`

**partialValues** — Values for a subset of leaf nodes of the resulting structure

Partial structure | []

Values for a subset of leaf nodes of the resulting structure, specified as a partial structure or empty array. Each field that you specify in a partial structure must match the data attributes of the corresponding bus element exactly. For details, see “Create Partial Structures for Initialization”.

Use an empty matrix [] when you use the `dims` argument and want to use ground values for all of the nodes in the resulting structure.

Data Types: `struct`
**dims — Dimensions of the resulting structure**
vector

Dimensions of the resulting structure, specified as a vector. Each dimension element must be an integer that is greater than or equal to 1. If you specify `partialValues`, then each dimension element in `dims` must be greater than or equal to its corresponding dimension element in the partial structure.

To use ground values, use an empty matrix ([]) for `partialValues`.

Data Types: double

**portHandles — Handles of bus signal ports**
array

Handles of bus signal ports, specified as an array. If you use the `partialStructures` argument, then the number of port handles that you specify in `portHandles` must be the same as the number of partial structures.

Data Types: double

**partialStructures — Partial structures**
cell array

Partial structures specified as a cell array. The number of port handles that you specify in `portHandles` must be the same as the number of partial structures.

Data Types: cell

**busObjectNames — Bus object names**
cell array

Bus object names, specified as a cell array.

Data Types: cell

### Output Arguments

**structFromBus — Bus signal hierarchy and attributes**
MATLAB structure

Bus signal hierarchy and attributes, returned as a MATLAB structure.
The dimensions of `structFromBus` depend on the input arguments:

- If you specify only `busSource`, then the dimension is 1.
- If you also specify `partialValues`, then the dimensions match the dimensions of `partialValues`.
- If you specify the `dims` argument, then the dimensions match the dimensions of `dims`.

`structsForBuses — Structures having the same hierarchy and attributes as bus signals  
cell array  
`  
Structures with the same hierarchy and attributes as bus signals, returned as a cell array of structures of data with same hierarchy and attributes as a bus signals that you specify with an array of port handles. The cell array of structures uses ground values of the bus signals.

The dimensions of `StructsForBuses` depend on the input arguments:

- If you specify only `portHandles`, then the dimension is 1.
- If you also specify `partialStructures`, then the dimensions match the dimensions of `partialStructures`.

**Tips**

- If you use the `Simulink.Bus.createMATLABStruct` function repeatedly for the same model (for example, in a loop in a script), you can improve performance by avoiding multiple model compilations. For improved speed, put the model in `compile` before using the function multiple times. For example, to put the `vdp` model in `compile`, use this command:

  \[ \text{sys}, \text{x0}, \text{str}, \text{ts} = \text{vdp}([],[],[],'compile') \]

  After you create the MATLAB structure, terminate the compile. For example:

  \[ \text{vdp}([],[],[],'term') \]

- You can use the Bus Editor to invoke the `Simulink.Bus.createMATLABStruct` function. Use one of these approaches:

  - Select the **File > Create a MATLAB structure** menu item.
• Select the bus object for which you want to create a full MATLAB structure. Then, in the toolbar, click the **Create a MATLAB structure** button (겠습니다).

You can then edit the MATLAB structure in the MATLAB Editor and evaluate the code to create or update the values in this structure.

• You can use the `Simulink.Bus.createMATLABStruct` function to specify the initial value of the output of a referenced model. For details, see the “Referenced Model: Setting Initial Value for Bus Output” section of the Detailed Workflow for Managing Data with Model Reference example.

## See Also

**Blocks**
Bus Assignment | Bus Creator | Bus to Vector

**Classes**
`Simulink.Bus` | `Simulink.BusElement`

**Functions**

**Topics**
“Types of Composite Signals”
“Specify Initial Conditions for Bus Signals”

**Introduced in R2010a**
Simulink.Bus.createObject

**Package:** Simulink.Bus

Create bus objects from blocks or MATLAB structures

**Syntax**

```matlab
busInfo = Simulink.Bus.createObject(model,blocks)
busInfo = Simulink.Bus.createObject(struct)
busInfo = Simulink.Bus.createObject(____,file)
busInfo = Simulink.Bus.createObject(____,format)
```

**Description**

`busInfo = Simulink.Bus.createObject(model,blocks)` creates `Simulink.Bus` objects for specified blocks and returns information about the created bus objects. These bus objects go in the MATLAB base workspace or, if available, the data dictionary used by the model.

`busInfo = Simulink.Bus.createObject(struct)` creates bus objects from a structure that can contain MATLAB timeseries, MATLAB timetable, and `matlab.io.datastore.SimulationDatastore` objects or a numeric structure. These bus objects go in the MATLAB base workspace or, if available, the data dictionary used by the model.

`busInfo = Simulink.Bus.createObject(____,file)` saves the bus objects in a MATLAB file that contains a cell array of cell arrays. Each subordinate cell array represents a bus object and contains this data:

- Bus name
- Header file
- Description
- Data scope
- Alignment
• Elements

The elements field is an array containing this data for each element:

• Element name
• Dimensions
• Data type
• Sample time
• Complexity
• Dimensions mode
• Minimum
• Maximum
• Units
• Description

`busInfo = Simulink.Bus.createObject(___,'format')` saves the bus objects in a file that contains either a cell array of bus information, or the bus objects themselves.

## Examples

### Create a Bus Object from a Bus Creator Block

Open the model.

`open_system('busdemo')`

Create a bus object that corresponds with the bus created by the Bus Creator2 block.

`bus2Info = Simulink.Bus.createObject...('busdemo','busdemo/Bus Creator2')`

### Create and Save Bus Objects to a File

Create a bus object from two Bus Creator blocks, using block handles to specify the blocks. Assign the block handles to variables and use the variables in a vector to specify the blocks used for creating the bus object.
Open the model.

```matlab
open_system('busdemo')
```

In the Simulink Editor, select the Bus Creator2 block.

In MATLAB, assign the block handle to a variable.

```matlab
bc2 = gcbh;
```

In the Simulink Editor, select the Bus Creator block.

In MATLAB, assign the block handle to a variable.

```matlab
bc1 = gcbh;
```

To create a bus object, use the block handle variables in a vector.

```matlab
busInfo = Simulink.Bus.createObject
('busdemo', [bc2 bc1], 'busdemo_busobject')
```

## Input Arguments

- **model** — Model name or handle  
  character vector

  Model name or handle, specified as a character vector.

- **blocks** — Blocks to create bus objects for  
  character vector | cell array of block path names | vector of block handles

  Blocks to create bus object for, specified as a character vector, cell array of block path names, or vector of block handles.

  - For just one block, specify the full path name of the block.
  - For multiple blocks, specify either a cell array of block path names or a vector of block handles.
  - If you specify a Bus Creator block that is at the highest level of a bus hierarchy, the function creates bus objects for all buses in the hierarchy.
**struct — Structure used to create bus objects**
structure of MATLAB timeseries, MATLAB timetable, and matlab.io.datastore.SimulationDatastore objects | numeric structure

Structure used to create bus objects, specified as a structure that can contain MATLAB timeseries, MATLAB timetable, and matlab.io.datastore.SimulationDatastore objects or a numeric structure.

**file — MATLAB file used to save bus object definitions**
character vector

MATLAB file used to save bus object definitions, specified as a character vector. The file name must be unique.

**format — Format for storing bus objects in file**
'cell' (default) | 'object'

Format for storing bus objects in file, specified as either 'cell' or 'object'. The cell format is more compact, but the object format is easier to read.

**Output Arguments**

**busInfo — Bus information for specified blocks**
structure array

Bus information for specified blocks, returned as a structure array. Each element of the structure array corresponds to one block and contains these fields:

- block — Handle of the block
- busName — Name of the bus object associated with the block

**Tips**

If you specify a model name, the model must compile successfully before you use the Simulink.Bus.createObject command.
Compatibility Considerations

Simulink.BusElement objects will no longer support the SampleTime property
Not recommended starting in R2019b

In a future release, the SampleTime property of Simulink.BusElement objects will be removed.

To prepare:

• When defining BusElement objects, use the default SampleTime value (-1).
• To specify the sample time for an element of a bus, use the SampleTime block parameter of corresponding blocks. For example, you can use In Bus Element, Out Bus Element, and Signal Specification blocks to specify sample time.

See Also
Bus Creator | Simulink.Bus | Simulink.Bus.cellToObject |
Simulink.Bus.createMATLABStruct | Simulink.Bus.objectToCell |
Simulink.BusElement

Topics
“Specify Bus Properties with Bus Objects”
“Create Bus Objects Programmatically”
“Create and Specify Bus Objects”
“Save Bus Objects”

Introduced before R2006a
Simulink.Bus.objectToCell

**Package:** Simulink.Bus

Use bus objects to create cell array containing bus information

**Syntax**

```matlab
cells = Simulink.Bus.objectToCell(busNames)
cells = Simulink.Bus.objectToCell(busNames,scope)
```

**Description**

`cells = Simulink.Bus.objectToCell(busNames)` creates a cell array of bus information from a set of bus objects in the MATLAB base workspace. The cell array contains subordinate cell arrays that define each bus object. The order of elements in the output cell array corresponds to the order of names in the input cell array.

`cells = Simulink.Bus.objectToCell(busNames,scope)` creates a cell array of bus information from a set of bus objects in a data dictionary. The cell array contains subordinate cell arrays that define each bus object. The order of elements in the output cell array corresponds to the order of names in the input cell array.

**Examples**

**Create Cell Array Containing Bus Object Information**

Use the `Simulink.Bus.objectToCell` function to create a cell array of information about bus objects in the base workspace.

Open a model that has two bus objects defined, CONTROL and MAIN.

```matlab
open_system(fullfile(matlabroot,'examples','simulink','ex_bus_object_tutorial_using_objects'))
```
Create a cell array of information about the CONTROL bus object.

```matlab
cells = Simulink.Bus.objectToCell({'CONTROL'});
cells{1}
```

```
ans =
1x6 cell array
{'CONTROL'}    {0x0 char}    {0x0 char}    {'Auto'}    {'-1'}    {2x1 cell}
```

### Input Arguments

**busNames — Names of bus objects**
cell array of bus object names

Bus objects for which to create cell arrays of bus object information, specified as a cell array. Specify the bus object names as character vectors. If `busNames` is empty, the function converts all bus objects in the base workspace or data dictionary.

Example: `cells = Simulink.Bus.objectToCell({'busObject'})`
**scope** — Data dictionary that contains bus objects
Simulink.data.Dictionary object

Data dictionary, specified as a Simulink.data.Dictionary object. Before you use this function, represent the dictionary with a Simulink.data.Dictionary object by using, for example, the Simulink.data.dictionary.create or Simulink.data.dictionary.open function.

If scope is empty, the function uses the MATLAB base workspace as the source of the bus objects.

Example: `cells = Simulink.Bus.objectToCell({},dataDictionaryObject)`

## Output Arguments

**cells** — Bus object information
cell array of cell arrays

Bus object information, specified as a cell array of cell arrays. Each subordinate cell array contains this bus object information:

1. Bus name
2. Header file
3. Description
4. Data scope
5. Alignment
6. Elements

The elements field is an array that contains this data for each bus element:

1. Bus element name
2. Dimensions
3. Data type
4. Sample time
5. Complexity
6. Sampling mode

The elements field also contains this data if available:
1 Dimensions mode
2 Minimum
3 Maximum
4 Units
5 Description

**Tips**

The inverse function is `Simulink.Bus.cellToObject`.

**Compatibility Considerations**

**Simulink.BusElement objects will no longer support the SampleTime property**  
*Not recommended starting in R2019b*

In a future release, the `SampleTime` property of `Simulink.BusElement` objects will be removed.

To prepare:

- When defining `BusElement` objects, use the default `SampleTime` value (`-1`).
- To specify the sample time for an element of a bus, use the `SampleTime` block parameter of corresponding blocks. For example, you can use In Bus Element, Out Bus Element, and Signal Specification blocks to specify sample time.

**Simulink.BusElement objects will no longer support the SamplingMode property**  
*Not recommended starting in R2016b*

In R2016b, the `SamplingMode` property of `Simulink.BusElement` objects was removed. Scripts that use the `SamplingMode` property of `Simulink.BusElement` objects continue to work. `Simulink.Bus.cellToObject` continues to require the `SamplingMode` field and `Simulink.Bus.objectToCell` continues to include the sampling mode in the output cell arrays.
In a future release, support for the SamplingMode property will be removed.

To specify whether a signal is sample-based or frame-based, define the sampling mode of input signals at the block level instead of at the signal level.

See Also

Classes
Simulink.Bus | Simulink.BusElement

Functions

Topics
“Specify Bus Properties with Bus Objects”
“Create Bus Objects Programmatically”
“Create and Specify Bus Objects”
“Save Bus Objects”

Introduced in R2007a
Simulink.Bus.save

Package: Simulink.Bus

Save bus objects in MATLAB file

Syntax

Simulink.Bus.save(fileName)
Simulink.Bus.save(fileName,format)
Simulink.Bus.save(fileName,format,busNames)

Description

Simulink.Bus.save(fileName) saves all bus objects (instances of Simulink.Bus class) that are in the MATLAB base workspace in a MATLAB file that contains a cell array of cell arrays. Each subordinate cell array represents a bus object and contains this data:

- Bus name
- Header file
- Description
- Data scope
- Alignment
- Elements

The elements field is an array containing this data for each element:

- Element name
- Dimensions
- Data type
- Sample time
- Complexity
- Dimensions mode
• Minimum
• Maximum
• Units
• Description

Simulink.Bus.save(fileName, format) saves the bus objects in a MATLAB file that contains either a cell array of bus information or the bus objects themselves.

Simulink.Bus.save(fileName, format, busNames) saves only those bus objects whose names appear in busNames.

**Examples**

**Save a Bus Object**

Use the Simulink.Bus.save function to save a bus object.

Define a cell array of bus object information.

```matlab
busCell = { ... 
    { ... 
    'myBusObj', ..., 
    'MyHeader.h', ..., 
    'My description', ..., 
    'Exported', ..., 
    '-1', ..., 
    {{'a',1,'double',[0.2 0],'real','Frame'}; ... 
    {'b',1,'double',[0.2 0],'real','Sample'}}, ... 
    }, ... 
};
```

Create `myBusObj` bus object from the cell array.

Simulink.Bus.cellToObject(busCell);

Save the bus object in the `BusCellFile1` file, in cell format.

```matlab
fileName = 'BusCellFile1';
Simulink.Bus.save(fileName);
```

Save the bus object in bus format.
Simulink.Bus.save('BusObjFile','object');

Save myBusObj in cell format in BusCellFile2.m.

Simulink.Bus.save('BusCellFile2','cell',{myBusObj});

### Input Arguments

**fileName** — File in which to store bus objects  
character vector

File in which to store bus objects, specified as a character vector.

**format** — Format for storing bus objects in file  
'cell' (default) | 'object'

Format for storing bus objects in file, specified as either 'cell' or 'object'. The cell format is more compact, but the object format is easier to read.

**busNames** — Bus objects to save  
cell array of bus objects

Bus objects to save, specified as a cell array of bus objects. Only the specified bus objects in the base workspace are saved.

### Tips

Executing a MATLAB file created by Simulink.Bus.save in cell array format calls Simulink.Bus.cellToObject to recreate the bus objects and returns the new bus objects in the cell array. To suppress the creation of bus objects, specify the optional argument 'false' when you execute the MATLAB file.
Compatibility Considerations

Simulink.BusElement objects will no longer support the SampleTime property
Not recommended starting in R2019b

In a future release, the SampleTime property of Simulink.BusElement objects will be removed.

To prepare:

• When defining BusElement objects, use the default SampleTime value (-1).
• To specify the sample time for an element of a bus, use the SampleTime block parameter of corresponding blocks. For example, you can use In Bus Element, Out Bus Element, and Signal Specification blocks to specify sample time.

See Also

Topics
“Specify Bus Properties with Bus Objects”
“Create Bus Objects Programmatically”
“Create and Specify Bus Objects”
“Save Bus Objects”

Introduced before R2006a
Simulink.clearIntEnumType

Delete enumeration classes defined by Simulink.defineIntEnumType

Syntax

Simulink.clearIntEnumType(typeName)
Simulink.clearIntEnumType()

Description

Simulink.clearIntEnumType(typeName) deletes a specific enumeration class that is defined by Simulink.defineIntEnumType. The function generates a warning if the class name is invalid or if a class cannot be deleted because instances of the class exist.

Simulink.clearIntEnumType() deletes all enumeration classes that are defined by Simulink.defineIntEnumType. The function generates a warning if a class cannot be deleted because instances of the class exist.

Examples

Delete a Specific Dynamic Enumerated Data Type

Define an enumeration type and confirm that it has been created.

Simulink.defineIntEnumType('myEnumType', {'e1', 'e2'}, [1 2]);
myResult = Simulink.findIntEnumType('myEnumType')

Delete the enumeration type that you created and confirm that it is no longer there.

Simulink.clearIntEnumType('myEnumType');
myResult = Simulink.findIntEnumType('myEnumType')
Delete All Dynamic Enumerated Data Types

Define two enumeration types and confirm that they have been created.

```matlab
Simulink.defineIntEnumType('myEnumType1', {'e1', 'e2'}, [1 2]);
Simulink.defineIntEnumType('myEnumType2', {'e3', 'e4'}, [3 4]);
myResult = Simulink.findIntEnumType()
```

Delete all enumeration types and confirm that no enumeration types exist.

```matlab
Simulink.clearIntEnumType();
myResult = Simulink.findIntEnumType()
```

### Input Arguments

typeName — Name of enumeration class
character vector or string

Name of a specific enumeration class that is defined by Simulink.defineIntEnumType, specified as a character vector or string.

Example: 'myEnumType'

Data Types: char | string

### See Also

```matlab
enumeration | Simulink.defineIntEnumType | Simulink.findIntEnumType
```

Introduced in R2018b
Simulink.createFromTemplate

Create model or project from template

Syntax

Simulink.createFromTemplate(templatename)

h = Simulink.createFromTemplate(templatename)

h = Simulink.createFromTemplate(templatename,Name,Value)

Description

Simulink.createFromTemplate(templatename) creates a model or a project from the template file specified by templatename.

h = Simulink.createFromTemplate(templatename) creates a model or a project from the template file and returns h, either a numeric model handle or a simulinkproject object.

h = Simulink.createFromTemplate(templatename,Name,Value) specifies additional options as one or more Name, Value pair arguments.

Examples

Create a Model From a Template

Simulink.createFromTemplate('simple_simulation.sltx')

Create a Project From a Template and Get the Handle

Create a project from a template, specify the name and root folder, and return the handle to the new project (a simulinkproject object) for manipulating it programmatically.
proj = Simulink.createFromTemplate('code_generation_example.sltx','Name','myProject','Folder','C:\Work\project1')

## Input Arguments

templatename — Template file name
character vector

Template file name, specified as a character vector. If the template is not on the MATLAB path, specify the fully-qualified path to the template file and *.sltx extension.

Example:

Data Types: char

## Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside quotes. You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

Example:

**Folder** — Project root folder
character vector

Project root folder, if creating a new project, specified as a character vector.

Data Types: char

**Name** — New model or project name
character vector

New model or project name, specified as a character vector.

Data Types: char

## Output Arguments

**h** — Handle
numeric handle | simulinkproject
Handle to the new model project, returned either as a numeric model handle or a `simulinkproject` object.

**See Also**
Simulink.defaultModelTemplate | Simulink.exportToTemplate | Simulink.findTemplates

**Topics**
“Create and Open Models”
“Create a Template from a Model”
“Using Templates to Create Standard Project Settings”

**Introduced in R2016a**
Simulink.data.assigninGlobal

Modify variable values in context of Simulink model

Syntax

Simulink.data.assigninGlobal(modelName, varName, varValue)

Description

Simulink.data.assigninGlobal(modelName, varName, varValue) assigns the value varValue to the variable or data dictionary entry varName in the context of the Simulink model modelName. assigninGlobal creates the variable or data dictionary entry if it does not already exist. The function operates in the Design Data section of the data dictionary that is linked to the target model or in the MATLAB base workspace if the target model is not linked to any data dictionary.

If the target model is linked to a data dictionary that references other dictionaries, assigninGlobal searches for varName in the entire dictionary hierarchy. If assigninGlobal does not find a matching entry, the function creates an entry in the dictionary that is linked to the target model.

Examples

Modify Variable in Model With or Without Data Dictionary

Create a variable myNewVariable with value 237 in the context of the Simulink model vdp.slx, which is not linked to any data dictionary. myNewVariable appears as a variable in the MATLAB base workspace.

Simulink.data.assigninGlobal('vdp', 'myNewVariable', 237)

Create a variable myNewEntry with value true in the context of the Simulink model sldemo_fuelsys_dd_controller.slx, which is linked to the data dictionary
sldemo_fuelsys_dd_controller.sldd. The entry myNewEntry appears in the Design Data section of the dictionary.

Simulink.data.assigninGlobal('sldemo_fuelsys_dd_controller',... 'myNewEntry',true)

Confirm the addition of myNewEntry to the data dictionary sldemo_fuelsys_dd_controller.sldd by viewing the dictionary in Model Explorer.

myDictionaryObj = Simulink.data.dictionary.open(... 'sldemo_fuelsys_dd_controller.sldd');
show(myDictionaryObj)

**Input Arguments**

**modelName** — Name of target Simulink model

character vector

Name of target Simulink model, specified as a character vector.

Example: 'myTestModel'

Data Types: char

**varName** — Name of target variable or data dictionary entry

character vector

Name of target variable or data dictionary entry, specified as a character vector.

Example: 'myTargetVariable'

Data Types: char

**varValue** — Value to assign to variable or data dictionary entry

MATLAB expression

Value to assign to variable or data dictionary entry, specified as a MATLAB expression that returns any valid data type or data dictionary content.

Example: 27.5
Example: myBaseWorkspaceVariable
Example: Simulink.Parameter
Tips

- `assigninGlobal` helps you transition Simulink models to using data dictionaries. You can use the function to assign values to model variables before and after linking a model to a data dictionary.

See Also

`Simulink.data.dictionary.open` | `Simulink.data.evalinGlobal` | `Simulink.data.existsInGlobal`

Topics

“Store Data in Dictionary Programmatically”
“What Is a Data Dictionary?”
“Considerations before Migrating to Data Dictionary”

 Introduced in R2015a
Simulink.data.dictionary.cleanupWorkerCache

Restore defaults after parallel simulation with data dictionary

Syntax

Simulink.data.dictionary.cleanupWorkerCache

Description

Simulink.data.dictionary.cleanupWorkerCache restores default settings after you have finished parallel simulation of a model that is linked to a data dictionary. Use this function in a spmd block, after you finish parallel simulation using parfor blocks, to restore default settings that were altered by the Simulink.data.dictionary.setupWorkerCache function.

During parallel simulation of a model that is linked to a data dictionary, you can allow each worker to access and modify the data in the dictionary independently of other workers. The function Simulink.data.dictionary.setupWorkerCache grants each worker a unique dictionary cache to allow independent access to the data, and the function Simulink.data.dictionary.cleanupWorkerCache restores cache settings to their default values.

You must have a Parallel Computing Toolbox license to perform parallel simulation using a parfor block.

Examples

Sweep Variant Control Using Parallel Simulation

To use parallel simulation to sweep a variant control (a Simulink.Parameter object whose value influences the variant condition of a Simulink.Variant object) that you
store in a data dictionary, use this code as a template. Change the names and values of the model, data dictionary, and variant control to match your application.

To sweep block parameter values or the values of workspace variables that you use to set block parameters, use Simulink.SimulationInput objects instead of the programmatic interface to the data dictionary. See “Optimize, Estimate, and Sweep Block Parameter Values”.

You must have a Parallel Computing Toolbox license to perform parallel simulation.

% For convenience, define names of model and data dictionary
model = 'mySweepMdl';
dd = 'mySweepDD.sldd';

% Define the sweeping values for the variant control
CtrlValues = [1 2 3 4];

% Grant each worker in the parallel pool an independent data dictionary
% so they can use the data without interference
spmd
    Simulink.data.dictionary.setupWorkerCache
end

% Determine the number of times to simulate
numberOfSims = length(CtrlValues);

% Prepare a nondistributed array to contain simulation output
simOut = cell(1,numberOfSims);

parfor index = 1: numberOfSims
% Create objects to interact with dictionary data
% You must create these objects for every iteration of the parfor-loop
dictObj = Simulink.data.dictionary.open(dd);
sectObj = getSection(dictObj,'Design Data');
entryObj = getEntry(sectObj,'MODE');
% Suppose MODE is a Simulink.Parameter object stored in the data dictionary

% Modify the value of MODE
temp = getValue(entryObj);
temp.Value = CtrlValues(index);
setValue(entryObj,temp);

% Simulate and store simulation output in the nondistributed array
simOut{index} = sim(model);
% Each worker must discard all changes to the data dictionary and close the dictionary when finished with an iteration of the parfor-loop
discardChanges(dictObj);
close(dictObj);
end

% Restore default settings that were changed by the function
% Simulink.data.dictionary.setupWorkerCache
% Prior to calling cleanupWorkerCache, close the model

spmd
    bdclose(model)
    Simulink.data.dictionary.cleanupWorkerCache
end

---

**Note** If data dictionaries are open, you cannot use the command Simulink.data.dictionary.cleanupWorkerCache. To identify open data dictionaries, use Simulink.data.dictionary.getOpenDictionaryPaths.

---

**See Also**
Simulink.data.dictionary.closeAll |
Simulink.data.dictionary.getOpenDictionaryPaths |
Simulink.data.dictionary.setupWorkerCache | parfor | spmd

**Topics**
“Store Data in Dictionary Programmatically”
“What Is a Data Dictionary?”
“Run Code on Parallel Pools” (Parallel Computing Toolbox)

**Introduced in R2015a**
Simulink.data.dictionary.closeAll

Close all connections to all open data dictionaries

Syntax

Simulink.data.dictionary.closeAll
Simulink.data.dictionary.closeAll(dictFileName)
Simulink.data.dictionary.closeAll(____,unsavedAction)

Description

Simulink.data.dictionary.closeAll attempts to close all connections to all data dictionaries that are open. For example, if you create objects, such as Simulink.data.Dictionary, that refer to a dictionary, that dictionary is open.

Some commands and functions, such as Simulink.data.dictionary.cleanupWorkerCache, cannot operate when dictionaries are open. It is a best practice to close each connection individually by using functions and methods such as the close method of a Simulink.data.Dictionary object. To find dictionaries that are open, use Simulink.data.dictionary.getOpenDictionaryPaths. However, you can use this function to close all connections to all dictionaries.

You can also use this function to close dictionaries in a shutdown script that is part of a project.

Simulink.data.dictionary.closeAll(dictFileName) closes all connections to the dictionary named dictFileName. If you open multiple dictionaries that use this file name (for example, if the dictionaries have different file paths), the function closes all connections to all of the dictionaries.

You cannot specify dictFileName as a full file path such as 'C:\temp\myDict.sldd'.

Simulink.data.dictionary.closeAll(____,unsavedAction) closes all connections to the target dictionaries by discarding or saving unsaved changes. You can choose whether to save or discard all changes to all of the target dictionaries.
Examples

Close All Connections to All Open Dictionaries

Discard any unsaved changes. All of the entries in the dictionaries revert to the last saved state.

Simulink.data.dictionary.closeAll('-discard')

Close All Connections to Single Data Dictionary

Open multiple connections to a data dictionary, make a change, and close all of the connections by discarding the unsaved change.

At the command prompt, open a data dictionary by creating a Simulink.data.Dictionary object that refers to the dictionary.

dictObj = Simulink.data.dictionary.open('sldemo_fuelsys_dd.sldd');

Display the dictionary in the Model Explorer

show(dictObj)

You now have two connections to this dictionary: The Simulink.data.Dictionary object and the Model Explorer.

Make a change to the dictionary by adding an entry.

dDataSectObj = getSection(dictObj,'Design Data');
addEntry(dDataSectObj,'myEntry',5.2);

The Simulink.data.dictionary.Section object dDataSectObj is a third connection to the dictionary.

Close the connections to the dictionary. Discard the unsaved change.

Simulink.data.dictionary.closeAll('sldemo_fuelsys_dd.sldd','-discard')

The dictionary no longer appears as a node in the Model Hierarchy pane of the Model Explorer. The Simulink.data.Dictionary object dictObj is disconnected from the
dictionary. You cannot interact with the dictionary by using the
Simulink.data.dictionary.Section object dDataSectObj.

Clear the objects that referred to the dictionary.

clear dictObj dDataSectObj

**Input Arguments**

dictFileName — File name of target data dictionary or dictionaries
character vector

File name of target data dictionary or dictionaries, specified as a character vector. Use
the file extension sldd.
Example: 'myDict.sldd'
Data Types: char

unsavedAction — Action for unsaved changes
'-discard' | '-save'

Action for unsaved changes, specified as '-discard' (to discard changes) or '-save'
(to save changes).

**Tips**

A data dictionary is open if any of these conditions are true:

- The dictionary appears as a node in the **Model Hierarchy** pane of the Model Explorer. To close this connection to the dictionary, right-click the node in Model Explorer and select **Close**. Alternatively, use the hide method of a Simulink.data.Dictionary object.
- You created an object of any of these classes that refer to the dictionary:
  - Simulink.data.Dictionary
  - Simulink.data.dictionary.Section
  - Simulink.data.dictionary.Entry
To close these connections to the dictionary, use the `close` method of the `Simulink.data.Dictionary` object or clear the object. Clear the `Simulink.data.dictionary.Section` and `Simulink.data.dictionary.Entry` objects.

- A model that is linked to the dictionary is open. To close this connection to the dictionary, close the model.

**See Also**

`Simulink.data.Dictionary` | `Simulink.data.dictionary.cleanupWorkerCache`  
| `Simulink.data.dictionary.getOpenDictionaryPaths`  
| `Simulink.data.dictionary.setupWorkerCache`

**Topics**

“Store Data in Dictionary Programmatically”

**Introduced in R2016a**
**Simulink.data.dictionary.create**

Create new data dictionary and create Simulink.data.Dictionary object

**Syntax**

```
dictionaryObj = Simulink.data.dictionary.create(dictionaryFile)
```

**Description**

`dictionaryObj = Simulink.data.dictionary.create(dictionaryFile)` creates a data dictionary file in your current working folder or in a file path you can specify in `dictionaryFile`. The function returns a Simulink.data.Dictionary object representing the new data dictionary.

**Examples**

**Create New Data Dictionary and Data Dictionary Object**

Create a data dictionary `myNewDictionary.sldd` in your current working folder and a Simulink.data.Dictionary object representing the new data dictionary. Assign the object to the variable `myDictionaryObj`.

```
myDictionaryObj = Simulink.data.dictionary.create('myNewDictionary.sldd')
```

```
myDictionaryObj =
    Dictionary with properties:
        DataSources: {0x1 cell}
        HasAccessToBaseWorkspace: 0
        EnableAccessToBaseWorkspace: 0
```
Input Arguments

dictionaryFile — Name of new data dictionary
character vector

Name of new data dictionary, specified as a character vector containing the file name and, optionally, path of the dictionary to create. If you do not specify a path, Simulink.data.dictionary.create creates the new data dictionary file in your working MATLAB folder. Simulink.data.dictionary.create also supports file paths specified relative to your working folder.

Example: 'myDictionary.sldd'
Example: 'C:\Users\jsmith\myDictionary.sldd'
Example: '..\myOtherDictionary.sldd'
Data Types: char

Output Arguments

dictionaryObj — Newly created data dictionary
Simulink.data.Dictionary object

Newly created data dictionary, returned as a Simulink.data.Dictionary object.

Alternatives

You can use the Simulink Editor to create a data dictionary and link it to a model. See “Migrate Single Model to Use Dictionary” for more information.

See Also

Simulink.data.Dictionary | Simulink.data.dictionary.open
Topics
“Store Data in Dictionary Programmatically”
“What Is a Data Dictionary?”

Introduced in R2015a
Simulink.data.dictionary.getOpenDictionaryPaths

Return file names and paths of open data dictionaries

Syntax

openDDs = Simulink.data.dictionary.getOpenDictionaryPaths
openDDs = Simulink.data.dictionary.getOpenDictionaryPaths(dictFileName)

Description

openDDs = Simulink.data.dictionary.getOpenDictionaryPaths returns the file names and paths of all data dictionaries that are open. For example, a data dictionary is open if you create objects, such as Simulink.data.Dictionary, that refer to the dictionary. If you open two or more dictionaries that have the same file name but different file paths, this function returns multiple file paths.

Before executing commands and functions that cannot operate when dictionaries are open, use this function to identify open dictionaries so that you can close them. For example, when you run parallel simulations as described in “Sweep Variant Control Using Parallel Simulation”, this function helps you identify open dictionaries before executing the command Simulink.data.dictionary.cleanupWorkerCache.

openDDs = Simulink.data.dictionary.getOpenDictionaryPaths(dictFileName) returns the file paths of data dictionaries that have the file name dictFileName. If you open two or more dictionaries that have the same file name but different file paths, you can use this syntax to return all of the file paths.

Examples

2-677
Identify and Close All Open Data Dictionaries

Open, identify, and close a data dictionary. After you close the connections to the dictionary, you can use commands and functions, such as `Simulink.data.dictionary.cleanupWorkerCache`, that cannot operate when dictionaries are open.

At the command prompt, open a data dictionary by creating a `Simulink.data.Dictionary` object that refers to the dictionary.

```matlab
dictObj = Simulink.data.dictionary.open('sldemo_fuelsys_dd.sldd');
```

Display the dictionary in the Model Explorer

```matlab
show(dictObj)
```

Identify all of the dictionaries that are open.

```matlab
openDDs = Simulink.data.dictionary.getOpenDictionaryPaths;
```

The file path of the dictionary that you opened, `sldemo_fuelsys_dd.sldd`, appears in the cell array of character vectors `openDDs`.

Close the connection from the Model Explorer to the dictionary.

```matlab
hide(dictObj)
```

The dictionary no longer appears as a node in the `Model Hierarchy` pane of the Model Explorer.

Close the connection from the `Simulink.data.Dictionary` object to the dictionary.

```matlab
close(dictObj)
clear dictObj
```

Input Arguments

dictFileName — File name of target data dictionary or dictionaries
character vector

File name of target data dictionary or dictionaries, specified as a character vector. Use the file extension `sldd`. 
Example: 'myDict.sldd'
Data Types: char

Output Arguments

openDDs — File names and paths of open data dictionaries
cell array of character vectors

File names and paths of open data dictionaries, returned as a cell array of character vectors.

Tips

A data dictionary is open if any of these conditions are true:

- The dictionary appears as a node in the Model Hierarchy pane of the Model Explorer. To close this connection to the dictionary, right-click the node in Model Explorer and select Close. Alternatively, use the hide method of a Simulink.data.Dictionary object.
- You created an object of any of these classes that refer to the dictionary:
  - Simulink.data.Dictionary
  - Simulink.data.dictionary.Section
  - Simulink.data.dictionary.Entry

To close these connections to the dictionary, use the close method of the Simulink.data.Dictionary object or clear the object. Clear the Simulink.data.dictionary.Section and Simulink.data.dictionary.Entry objects.
- A model that is linked to the dictionary is open. To close this connection to the dictionary, close the model.

See Also

Simulink.data.Dictionary | Simulink.data.dictionary.cleanupWorkerCache |
Simulink.data.dictionary.closeAll |
Simulink.data.dictionary.setupWorkerCache
Topics
“Store Data in Dictionary Programmatically”

Introduced in R2016a
Simulink.data.dictionary.open

Open data dictionary for editing

Syntax

dictionaryObj = Simulink.data.dictionary.open(dictionaryFile)

Description

dictionaryObj = Simulink.data.dictionary.open(dictionaryFile) opens the specified data dictionary and returns a Simulink.data.Dictionary object representing an existing data dictionary identified by its file name and, optionally, file path with dictionaryFile.

Make sure any dictionaries referenced by the target dictionary are on the MATLAB path.

Examples

Open Existing Data Dictionary

Open the data dictionary myDictionary_ex_API.sldd and represent it with a Simulink.data.Dictionary object named myDictionaryObj.

myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd')

myDictionaryObj =

    Dictionary with properties:
        DataSources: {'myRefDictionary_ex_API.sldd'}
Input Arguments

dictionaryFile — Target data dictionary
character vector

Target data dictionary, specified as a character vector containing the file name and, optionally, path of the dictionary. If you do not specify a path, Simulink.data.dictionary.open searches the MATLAB path for the specified file. Simulink.data.dictionary.open also supports paths specified relative to the MATLAB working folder.

Example: 'myDictionary_ex_API.sldd'
Example: 'C:\Users\jsmith\myDictionary_ex_API.sldd'
Example: '..\myOtherDictionary.sldd'
Data Types: char

See Also
Simulink.data.Dictionary | Simulink.data.dictionary.create | show

Topics
“Store Data in Dictionary Programmatically”

Introduced in R2015a
Simulink.data.dictionary.setupWorkerCache

Enable parallel simulation with data dictionary

Syntax

Simulink.data.dictionary.setupWorkerCache

Description

Simulink.data.dictionary.setupWorkerCache prepares the workers in a parallel pool for simulating a model that is linked to a data dictionary. Use this function in a spmd block, prior to starting a parfor block, to provide the workers in a parallel pool a way to safely interact with a single data dictionary.

During parallel simulation of a model that is linked to a data dictionary, you can allow each worker to access and modify the data in the dictionary independently of other workers. Simulink.data.dictionary.setupWorkerCache temporarily provides each worker in the pool with its own data dictionary cache, allowing the workers to use the data in the dictionary without permanently changing it.

You must have a Parallel Computing Toolbox license to perform parallel simulation using a parfor block.

Examples

Sweep Variant Control Using Parallel Simulation

To use parallel simulation to sweep a variant control (a Simulink.Parameter object whose value influences the variant condition of a Simulink.Variant object) that you store in a data dictionary, use this code as a template. Change the names and values of the model, data dictionary, and variant control to match your application.

To sweep block parameter values or the values of workspace variables that you use to set block parameters, use Simulink.SimulationInput objects instead of the
programmatic interface to the data dictionary. See “Optimize, Estimate, and Sweep Block Parameter Values”.

You must have a Parallel Computing Toolbox license to perform parallel simulation.

% For convenience, define names of model and data dictionary
model = 'mySweepMdl';
dd = 'mySweepDD.sldd';

% Define the sweeping values for the variant control
CtrlValues = [1 2 3 4];

% Grant each worker in the parallel pool an independent data dictionary % so they can use the data without interference
spmd
    Simulink.data.dictionary.setupWorkerCache
end

% Determine the number of times to simulate
numberOfSims = length(CtrlValues);

% Prepare a nondistributed array to contain simulation output
simOut = cell(1,numberOfSims);

parfor index = 1:numberOfSims
    % Create objects to interact with dictionary data
    % You must create these objects for every iteration of the parfor-loop
dictObj = Simulink.data.dictionary.open(dd);
sectObj = getSection(dictObj,'Design Data');
entryObj = getEntry(sectObj,'MODE');
    % Suppose MODE is a Simulink.Parameter object stored in the data dictionary

    % Modify the value of MODE
temp = getValue(entryObj);
temp.Value = CtrlValues(index);
setValue(entryObj,temp);

    % Simulate and store simulation output in the nondistributed array
simOut{index} = sim(model);

    % Each worker must discard all changes to the data dictionary and % close the dictionary when finished with an iteration of the parfor-loop
discardChanges(dictObj);
close(dictObj);
end

% Restore default settings that were changed by the function
% Simulink.data.dictionary.setupWorkerCache
% Prior to calling cleanupWorkerCache, close the model

spmd
    bdclose(model)
    Simulink.data.dictionary.cleanupWorkerCache
end

Note If data dictionaries are open, you cannot use the command
Simulink.data.dictionary.cleanupWorkerCache. To identify open data
dictionaries, use Simulink.data.dictionary.getOpenDictionaryPaths.

See Also
Simulink.data.dictionary.cleanupWorkerCache | parfor | spmd

Topics
“Store Data in Dictionary Programmatically”
“What Is a Data Dictionary?”
“Run Code on Parallel Pools” (Parallel Computing Toolbox)

Introduced in R2015a
Simulink.data.evalinGlobal

Evaluate MATLAB expression in context of Simulink model

Syntax

returnValue = Simulink.data.evalinGlobal(modelName,expression)

Description

returnValue = Simulink.data.evalinGlobal(modelName,expression) evaluates the MATLAB expression expression in the context of the Simulink model modelName and returns the values returned by expression. evalinGlobal evaluates expression in the Design Data section of the data dictionary that is linked to the target model or in the MATLAB base workspace if the target model is not linked to any data dictionary.

Examples

Evaluate MATLAB Expression in Model With or Without Data Dictionary

Evaluate the MATLAB expression myNewVariable = 237; in the context of the model vdp, which is not linked to any data dictionary. myNewVariable appears as a variable in the MATLAB base workspace.

Simulink.data.evalinGlobal('vdp','myNewVariable = 237;')

Evaluate the MATLAB expression myNewEntry = true; in the context of the model sldemo_fuelsys_dd_controller, which is linked to the data dictionary sldemo_fuelsys_dd_controller.sldd. myNewEntry appears as an entry in the Design Data section of the dictionary.

Simulink.data.evalinGlobal('sldemo_fuelsys_dd_controller',... 'myNewEntry = true;')
Confirm the creation of the entry `myNewEntry` in the data dictionary `sldemo_fuelsys_dd_controller.sldd` by viewing the dictionary in Model Explorer.

```matlab
myDictionaryObj = Simulink.data.dictionary.open('sldemo_fuelsys_dd_controller.sldd');
show(myDictionaryObj)
```

### Input Arguments

- **modelName** — Name of target Simulink model  
  character vector  
  Name of target Simulink model, specified as a character vector.  
  Example: `'myTestModel'`  
  Data Types: `char`

- **expression** — MATLAB expression to evaluate  
  character vector  
  MATLAB expression to evaluate, specified as a character vector.  
  Example: `'a = 5.3'`  
  Example: `'whos'`  
  Example: `'CurrentSpeed.Value = 290.73'`  
  Data Types: `char`

### Output Arguments

- **returnValue** — Value returned by specified expression  
  valid entry or variable value  
  Value returned by the specified MATLAB expression.
**Tips**

- `evalinGlobal` helps you transition Simulink models to the use of data dictionaries. You can use the function to manipulate model variables before and after linking a model to a data dictionary.

**See Also**

`Simulink.data.assigninGlobal` | `Simulink.data.existsInGlobal` | `evalin`

**Topics**

“Store Data in Dictionary Programmatically”

**Introduced in R2015a**
Simulink.data.existsInGlobal

Check existence of variable in context of Simulink model

Syntax

varExists = Simulink.data.existsInGlobal(modelName,varName)

Description

`varExists = Simulink.data.existsInGlobal(modelName,varName)` returns an indication of the existence of a variable or data dictionary entry `varName` in the context of the Simulink model `modelName`. `Simulink.data.existsInGlobal` searches the Design Data section of the data dictionary that is linked to the target model or the MATLAB base workspace if the target model is not linked to any data dictionary.

Examples

Determine Existence of Variable in Model With or Without Data Dictionary

Determine the existence of a variable `PressVect` in the context of the Simulink model `vdp.slx`, which is not linked to any data dictionary.

Simulink.data.existsInGlobal('vdp','PressVect')

ans =

0

Because `vdp.slx` is not linked to any data dictionary, `existsInGlobal` searches only in the MATLAB base workspace for `PressVect`.

Determine the existence of a variable `PressVect` in the context of the Simulink model `sldemo_fuelsys_dd_controller.slx`, which is linked to the data dictionary `sldemo_fuelsys_dd_controller.sldd`. 
Simulink.data.existsInGlobal('sldemo_fuelsys_dd_controller','PressVect')
ans =
    1

Because sldemo_fuelsys_dd_controller.slx is linked to the data dictionary
sldemo_fuelsys_dd_controller.sldd, existsInGlobal searches for PressVect
only in the Design Data section of the dictionary.

Input Arguments

modelName — Name of target Simulink model
character vector

Name of target Simulink model, specified as a character vector.
Example: 'myTestModel'
Data Types: char

varName — Name of target variable or data dictionary entry
character vector

Name of target variable or data dictionary entry, specified as a character vector.
Example: 'myTargetVariable'
Data Types: char

Output Arguments

varExists — Indication of existence of target variable or data dictionary entry
1|0

Indication of existence of target variable or data dictionary entry, returned as 1 to
indicate existence or 0 to indicate absence.
Tips

- `existsInGlobal` helps you transition Simulink models to the use of data dictionaries. You can use the function to find model variables before and after linking a model to a data dictionary.

Alternatives

You can use Model Explorer to search a data dictionary or any workspace for entries or variables.

See Also

`Simulink.data.assigninGlobal | Simulink.data.evalinGlobal | exist`

Topics

“Store Data in Dictionary Programmatically”

Introduced in R2015a
Simulink.data.getEnumTypeInfo

Get information about enumerated data type

Syntax

information = Simulink.data.getEnumTypeInfo(enumTypeName, infoRequest)

Description

information = Simulink.data.getEnumTypeInfo(enumTypeName, infoRequest) returns information about an enumerated data type enumTypeName.

Use this function only to return information about an enumerated data type. To customize an enumerated data type, for example, by specifying a default enumeration member or by controlling the scope of the type definition in generated code, see “Customize Simulink Enumeration”.

Examples

Return Default Value of Enumerated Data Type

Get the default enumeration member of an enumerated data type LEDcolor. Suppose LEDcolor defines two enumeration members, GREEN and RED, and uses GREEN as the default member.

Simulink.data.getEnumTypeInfo('LEDcolor','DefaultValue')

ans =

GREEN
Get Scope of Enumerated Data Type Definition in Generated Code

For an enumerated data type LEDcolor, find out if generated code exports or imports the definition of the type to or from a header file.

```matlab
Simulink.data.getEnumTypeInfo('LEDcolor','DataScope')
Simulink.data.getEnumTypeInfo('LEDcolor','HeaderFile')
```

ans =
Auto

ans =

Because DataScope is 'Auto' and HeaderFile is empty, generated code defines the enumerated data type LEDcolor in the header file `model_types.h` where `model` is the name of the model used to generate code.

### Input Arguments

- **enumTypeName** — Name of target enumerated data type
default: character vector

Name of the target enumerated data type, specified as a character vector.

Example: 'myFirstEnumType'

Data Types: char

- **infoRequest** — Information to return
default: valid character vector

Information to return, specified as one of the character vector options in the table.

<table>
<thead>
<tr>
<th>Specified value</th>
<th>Information returned</th>
<th>Example return value</th>
</tr>
</thead>
<tbody>
<tr>
<td>'DefaultValue'</td>
<td>The default enumeration member, returned as an instance of the enumerated data type.</td>
<td>enumMember1</td>
</tr>
<tr>
<td>Specified value</td>
<td>Information returned</td>
<td>Example return value</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>-----------------------</td>
</tr>
<tr>
<td>'Description'</td>
<td>The custom description of this data type, returned as a character vector. Returns an empty character vector if a description was not specified for the type.</td>
<td>'My first enum type.'</td>
</tr>
<tr>
<td>'HeaderFile'</td>
<td>The name of the custom header file that defines the data type in generated code, returned as a character vector. Returns an empty character vector if a header file was not specified for the type.</td>
<td>'myEnumType.h'</td>
</tr>
<tr>
<td>'DataScope'</td>
<td>Indication whether generated code imports or exports the definition of the data type. A return value of 'Auto' indicates generated code defines the type in the header file model_types.h or imports the definition from the header file identified by HeaderFile. A return value of 'Exported' or 'Imported' indicates generated code exports or imports the definition to or from the header file identified by HeaderFile.</td>
<td>'Exported'</td>
</tr>
<tr>
<td>'StorageType'</td>
<td>The integer data type used by generated code to store the numeric values of the enumeration members, returned as a character vector. Returns 'int' if you did not specify a storage type for the enumerated type, in which case generated code uses the native integer type of the hardware target.</td>
<td>'int32'</td>
</tr>
<tr>
<td>'AddClassNameToEnumNames'</td>
<td>Indication whether generated code prefixes the names of enumeration members with the name of the data type. Returned as true or false.</td>
<td>true</td>
</tr>
</tbody>
</table>

**See Also**

Simulink.defineIntEnumType
Topics
“Customize Simulink Enumeration”
“Simulink Enumerations”

Introduced in R2014b
Simulink.data.isSupportedEnumClass

Determine whether an enumeration class is valid for Simulink

Syntax

valid = Simulink.data.isSupportedEnumClass(enumClass)

Description

valid = Simulink.data.isSupportedEnumClass(enumClass) returns true if enumClass is a valid enumeration class for Simulink.

Examples

Determine if Enumeration Class is Valid

Define an enumeration class of basic colors, then confirm that it is valid.

Simulink.defineIntEnumType('BasicColors', ...
    {'Red', 'Yellow', 'Blue'}, ...
    [0;1;2]);
Simulink.data.isSupportedEnumClass('BasicColors')

ans =

    logical

    1

Perform the same validity check by using the meta.class object as the input.

Simulink.data.isSupportedEnumClass(?BasicColors)

ans =

2-696
logical
1

Check if `Simulink.NumericType` is a valid enumeration class by using the `meta.class` object as the input.

```matlab
Simulink.data.isSupportedEnumClass('Simulink.NumericType')
```

```
ans =
      logical
       0
```

**Input Arguments**

`enumClass` — Enumeration class
character vector | `meta.class`

Enumeration class, specified as a character vector or as `meta.class` object.

Example: `'myEnumClass'` or `?myEnumClass`

**Output Arguments**

`valid` — Whether enumeration class is valid
logical

If the input value refers to a valid enumeration class for Simulink, `valid` is `true`, otherwise `valid` is `false`.

**See Also**

- `Simulink.clearIntEnumType` | `Simulink.data.getEnumTypeInfo`
- `Simulink.data.isSupportedEnumObject` | `Simulink.defineIntEnumType`
- `Simulink.findIntEnumType`
Topics

“Simulink Enumerations”

Introduced in R2014b
Simulink.data.isSupportedEnumObject

Determine whether an enumeration object is valid for Simulink

**Syntax**

```matlab
valid = Simulink.data.isSupportedEnumObject(enumObject)
```

**Description**

```matlab
valid = Simulink.data.isSupportedEnumObject(enumObject)
```
returns true if `enumObject` is a valid enumeration object for Simulink.

**Examples**

**Determine if Enumeration Object is Valid**

Define an enumeration class of basic colors.

```matlab
Simulink.defineIntEnumType('BasicColors', ...
    {'Red', 'Yellow', 'Blue'}, ...
    [0;1;2]);
```

Confirm that an instance of the enumeration class is valid.

```matlab
Simulink.data.isSupportedEnumObject(BasicColors.Yellow)
ans =
    logical
    1
```

Call the function with input that is not an enumeration object.

```matlab
Simulink.data.isSupportedEnumObject(5)
```
ans =
    logical
           0

**Input Arguments**

enumObject — Enumeration object

Instance of a defined enumeration, specified as an enumeration object.
Example: `myEnumObject`

**Output Arguments**

valid — Whether enumeration class is valid

logical

If the input value refers to a valid enumeration object for Simulink, `valid` is `true`, otherwise `valid` is `false`.

**See Also**

- `Simulink.clearIntEnumType`
- `Simulink.data.getEnumTypeInfo`
- `Simulink.data.isSupportedEnumClass`
- `Simulink.defineIntEnumType`
- `Simulink.findIntEnumType`

**Topics**

“Simulink Enumerations”

**Introduced in R2014b**
Simulink.defineIntEnumType

Define enumerated data type

Syntax

Simulink.defineIntEnumType(ClassName, CellOf Enums, IntValues)
Simulink.defineIntEnumType(ClassName, CellOf Enums, IntValues, 'Description', ClassDesc)
Simulink.defineIntEnumType(ClassName, CellOf Enums, IntValues, 'DefaultValue', DefValue)
Simulink.defineIntEnumType(ClassName, CellOf Enums, IntValues, 'DataScope', ScopeSelection)
Simulink.defineIntEnumType(ClassName, CellOf Enums, IntValues, 'HeaderFile', FileName)
Simulink.defineIntEnumType(ClassName, CellOf Enums, IntValues, 'AddClassNameToEnumNames', Flag)
Simulink.defineIntEnumType(ClassName, CellOf Enums, IntValues, 'StorageType', DataType)

Description

Simulink.defineIntEnumType(ClassName, CellOf Enums, IntValues) defines an enumeration named ClassName with enumeration values specified with CellOf Enums and underlying numeric values specified by IntValues.

Simulink.defineIntEnumType(ClassName, CellOf Enums, IntValues, 'Description', ClassDesc) defines the enumeration with a description (character vector).

Simulink.defineIntEnumType(ClassName, CellOf Enums, IntValues, 'DefaultValue', DefValue) defines a default value for the enumeration, which is one of the character vectors you specify for CellOf Enums.

Simulink.defineIntEnumType(ClassName, CellOf Enums, IntValues, 'DataScope', ScopeSelection) specifies whether the data type definition should be imported from, or exported to, a header file during code generation.
Simulink.defineIntEnumType(ClassName, CellOfEnums, IntValues, 'HeaderFile', FileName) specifies the name of a header file containing the enumeration class definition for use in code generated from a model.

Simulink.defineIntEnumType(ClassName, CellOfEnums, IntValues, 'AddClassNameToEnumNames', Flag) specifies whether the code generator applies the class name as a prefix to the enumeration values that you specify for CellOfEnums. For Flag, specify true or false. For example, if you specify true, the code generator would use BasicColors.Red instead of Red to represent an enumerated value.

Simulink.defineIntEnumType(ClassName, CellOfEnums, IntValues, 'StorageType', DataType) specifies the data type used to store the enumerations’ underlying integer values in code generated from a model.

**Input Arguments**

**ClassName**

The name of the enumerated data type.

**CellOfEnums**

A cell array of character vectors that defines the enumerations for the data type.

**IntValues**

An array of numeric values that correspond to enumerations of the data type.

'**Description**, ClassDesc'

Specifies a character vector that describes the enumeration data type.

'**DefaultValue**, DefValue'

Specifies the default enumeration value.

'**HeaderFile**, FileName'

Specifies a character vector naming the header file that is to contain the data type definition.
By default, the generated `#include` directive uses the preprocessor delimiter " instead of `< and >. To generate the directive `#include <myTypes.h>`, specify `FileName` as `'<myTypes.h>'`.

`'DataScope'`, `'Auto' | 'Exported' | 'Imported'`

Specifies whether the data type definition should be imported from, or exported to, a header file during code generation.

<table>
<thead>
<tr>
<th>Value</th>
<th>Action</th>
</tr>
</thead>
</table>
| Auto (default) | If no value is specified for `Headerfile`, export the type definition to `model_types.h`, where `model` is the model name.  
If a value is specified for `Headerfile`, import the data type definition from the specified header file. |
| Exported          | Export the data type definition to a header file.  
If no value is specified for `Headerfile`, the header file name defaults to `type.h`, where `type` is the data type name. |
| Imported          | Import the data type definition from a header file.  
If no value is specified for `Headerfile`, the header file name defaults to `type.h`, where `type` is the data type name. |

`'AddClassNameToEnumNames'`, `Flag`

A logical flag that specifies whether code generator applies the class name as a prefix to the enumerations.

`'StorageType'`, `DataType`

Specifies a character vector that identifies the data type used to store the enumerations’ underlying integer values in generated code. The following data types are supported: `'int8'`, `'int16'`, `'int32'`, `'uint8'`, or `'uint16'`. 
Examples

Assume an external data dictionary includes the following enumeration:

BasicColors.Red(0), BasicColors.Yellow(1), BasicColors.Blue(2)

Import the enumeration class definition into the MATLAB workspace while specifying int16 as the underlying integer data type for generated code:

```matlab
Simulink.defineIntEnumType('BasicColors', ...
    {'Red', 'Yellow', 'Blue'}, ...
    [0;1;2], ...
    'Description', 'Basic colors', ...
    'DefaultValue', 'Blue', ...
    'HeaderFile', 'mybasiccolors.h', ...
    'DataScope', 'Exported', ...
    'AddClassNameToEnumNames', true, ...
    'StorageType', 'int16');
```

See Also

enumeration

Topics

“Import Enumerations Defined Externally to MATLAB”  
“Define Simulink Enumerations”

Introduced in R2010b
Simulink.defaultModelTemplate

Set or get default model template

Syntax

Simulink.defaultModelTemplate(templatename)

templatepath = Simulink.defaultModelTemplate

Description

Simulink.defaultModelTemplate(templatename) sets the template file specified by templatename as the default model template to use for new models. This setting is persistent between Simulink sessions.

templatepath = Simulink.defaultModelTemplate gets the full path to the current default model template.

Examples

Set the default model template

Simulink.defaultModelTemplate('simple_simulation.sltx')

Get the default model template

mydefaulttemplate = Simulink.defaultModelTemplate

Clear and restore the default model template

Use set_param to set a root block diagram parameter. This clears the default template so that new models will inherit this property of the root block diagram, and warns.
set_param(0,'StopTime','99');

Restore the default template.

Simulink.defaultModelTemplate('$restore');

**Input Arguments**

*templatename* — Template file name
character vector

Template file name, specified as a character vector. If the template is not on the MATLAB path, specify the fully-qualified path to the template file and *slt* extension.

Example: \Home\username\Documents\MATLAB\template.sltx

Data Types: char

**Output Arguments**

*templatepath* — Template path
character vector

Template path, specified as a character vector, showing the full path to the current default model template.

**See Also**

Simulink.createFromTemplate | Simulink.exportToTemplate | Simulink.findTemplates | new_system

**Topics**

“Create and Open Models”
“Create a Template from a Model”
“Using Templates to Create Standard Project Settings”

**Introduced in R2016b**
Simulink.exportToTemplate

Create template from model or project

Syntax

templatefile = Simulink.exportToTemplate(obj,templatename)
templatefile = Simulink.exportToTemplate(obj,templatename,
Name,Value)

Description

templatefile = Simulink.exportToTemplate(obj,templatename) creates a template file (templatename.sltx) from a model or project specified by obj.

If you have project templates created in R2014a or earlier (.zip files), use Simulink.exportToTemplate to upgrade them to .sltx files, then you can use them in the start page.

templatefile = Simulink.exportToTemplate(obj,templatename,
Name,Value) specifies additional template options as one or more Name, Value pair arguments.

Examples

Create a Template From a Model

Open the vdp model and create a template from it.

vdp
myvdptemplate = Simulink.exportToTemplate(bdroot,'vdptemplate')
Create a Template From a Model and Specify Description

Open the *vdp* model and create a template from it, specifying a description.

```matlab
vdp
myvdptemplate = Simulink.exportToTemplate(bdroot,'vdptemplate','Description','Use this template to create a vdp model')
```

**Input Arguments**

- **obj** — Model, library, or project
  
  Character vector | numeric handle | `slproject.ProjectManager`
  
  Model, library, or project, specified by name or numeric handle, or a `slproject.ProjectManager` object returned by the `simulinkproject` function.
  
  **Data Types:** double | char

- **templatename** — Template file name
  
  Character vector
  
  Template file name, specified as a character vector that can optionally include the fully-qualified path to a template file and *.*sltx extension.
  
  **Data Types:** char

**Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `Title, 'My Project Template'`

- **Group** — Group of template
  
  Character vector
  
  Group of template, specified as a character vector. On the Start Page, templates are shown under group headings.
  
  Example: `'Simscape'`
  
  **Data Types:** char
**Author — Author of template**
character vector

Author of template, specified as a character vector.
Data Types: char

**Description — Description of template**
character vector

Description of template, specified as a character vector.
Data Types: char

**ThumbnailFile — Thumbnail image file name**
character vector

Thumbnail image file name, specified as a character vector.
Data Types: char

**Title — Title of model or project template**
character vector

Title of template, specified as a character vector. On the Start Page, the templates titles are shown on the tiles. The title can be different from the file name, and you can use any characters in the title. The default value is the name of the model or project.

Example: 'My Project Template'
Data Types: char

**Output Arguments**

**templatefile — Template file**
character vector

Template file, returned as `templatename.sltx` file.

**See Also**

`Simulink.createFromTemplate` | `Simulink.defaultModelTemplate` | `Simulink.findTemplates`
Topics

“Create and Open Models”
“Create a Template from a Model”
“Using Templates to Create Standard Project Settings”

Introduced in R2016a
Simulink.exportToVersion

Export model or library for use in previous version of Simulink

Syntax

exported_file = Simulink.exportToVersion(modelname,target_filename,version)
exported_file = Simulink.exportToVersion(modelname,target_filename,version,Name,Value)

Description

exported_file = Simulink.exportToVersion(modelname,target_filename,version) exports the model or library modelname to a file named target_filename in a format that the specified previous Simulink version can load.

If the system contains functionality not supported by the specified Simulink software version, the command removes the functionality and replaces any unsupported blocks with empty masked subsystem blocks colored yellow. As a result, the converted system may generate different results.

The save_system ExportToVersion option is a legacy option for this functionality that is also supported.

exported_file = Simulink.exportToVersion(modelname,target_filename,version,Name,Value) specifies additional options as one or more Name, Value pair arguments.

Examples

Export a Model to a Previous Version

Get the current top-level system and export it.
Simulink.exportToVersion(bdroot,'mymodel.slx','R2014b');

Export a Model to a Previous Version and Break Links

Get the current top-level system and export it, replacing links to library blocks with copies of the library blocks in the saved file.

Simulink.exportToVersion(bdroot,'mymodel.slx','R2014b','BreakUserLinks',true);

Input Arguments

**modelname** — Model to export
character vector

Model to export, specified as a character vector, without any file extension. The model must be loaded and unmodified. The target file must not be the same as the model file.

Data Types: char

**target_filename** — Exported file name
character vector

Exported file name, specified as a character vector. The target file must not be the same as the model file.

Example: 'mymodel.slx'

Data Types: char

**version** — MATLAB release name

'R2013B' | 'R2014A_MDL' | 'R2016B_SLX' | ...

MATLAB release name, specified as a character vector, which specifies a previous Simulink version. Simulink.exportToVersion exports the system to a format that the specified previous Simulink version can load. You cannot export to your current version. These version names are not case sensitive.

You can specify model file format as SLX or MDL using the suffix _MDL or _SLX. If you do not specify a format, you export your default model file format.
If you use the Export to Previous Version dialog box instead of Simulink.exportToVersion, then the **Save as type** list supports 7 years of previous releases.

Example: 'R2015B'

Data Types: char

**Name-Value Pair Arguments**

Specify optional comma-separated pairs of **Name, Value** arguments. **Name** is the argument name and **Value** is the corresponding value. **Name** must appear inside quotes. You can specify several name and value pair arguments in any order as **Name1, Value1, ..., NameN, ValueN**.

Example:

**AllowPrompt — Allow prompt or message dialog box**

false (default) | true | 'on' | 'off'

Allow prompt or message dialog box, specified by a logical value that indicates whether to display any output prompt or message in a dialog box or only messages at the command line. For example, prompts to make files writable, or messages about exported versions. If you want to allow prompts, then set to **true**. or **on**.

**BreakUserLinks — Break user-defined links**

false (default) | true | 'on' | 'off'

Break user-defined links, specified by a logical value that indicates whether the function replaces links to user-defined library blocks with copies of the library blocks in the saved file.

**BreakToolboxLinks — Break all toolbox links**

false (default) | true | 'on' | 'off'

Break all toolbox links, specified by a logical value that indicates whether the function replaces links to built-in MathWorks library blocks with copies of the library blocks in the saved file. The 'BreakToolboxLinks' option affects Simulink library blocks and blocks from any other libraries supplied with MathWorks toolboxes or blocksets.

**Note** The 'BreakToolboxLinks' option can result in compatibility issues when upgrading to newer versions of Simulink software. For example:
• Any masks on top of library links to Simulink S-functions will not upgrade to the new version of the S-function.
• Any library links to masked subsystems in a Simulink library will not upgrade to the new subsystem behavior.
• Any broken links prevent the automatic library forwarding mechanism from upgrading the link.

If you have saved a model with broken links to built-in libraries, use the Upgrade Advisor to scan the model for out-of-date blocks and upgrade the Simulink blocks to their current versions.

Output Arguments

exported_file — Exported file
character vector

Exported file, returned in the format that the specified previous Simulink version can load.

See Also

save_system

Topics

“Save a Model”

Introduced in R2016a
Simulink.fileGenControl

Specify root folders for files generated by diagram updates and model builds

Syntax

cfg = Simulink.fileGenControl('getConfig')
Simulink.fileGenControl(Action,Name,Value)

Description

cfg = Simulink.fileGenControl('getConfig') returns a handle to an instance of the Simulink.FileGenConfig object, which contains the current values of these file generation control parameters:

- CacheFolder - Specifies the root folder for model build artifacts that are used for simulation, including Simulink® cache files.
- CodeGenFolder - Specifies the root folder for code generation files.
- CodeGenFolderStructure - Controls the folder structure within the code generation folder.

To get or set the parameter values, use the Simulink.FileGenConfig object.

These Simulink preferences determine the initial parameter values for the MATLAB session:

- Simulation cache folder - CacheFolder
- Code generation folder - CodeGenFolder
- Code generation folder structure - CodeGenFolderStructure

Simulink.fileGenControl(Action,Name,Value) performs an action that uses the file generation control parameters of the current MATLAB session. Specify additional options with one or more name, value pair arguments.
Examples

Get File Generation Control Parameter Values

To obtain the file generation control parameter values for the current MATLAB session, use getConfig.

cfg = Simulink.fileGenControl('getConfig');

myCacheFolder = cfg.CacheFolder;
myCodeGenFolder = cfg.CodeGenFolder;
myCodeGenFolderStructure = cfg.CodeGenFolderStructure;

Set File Generation Control Parameters by Using Simulink.FileGenConfig Object

To set the file generation control parameter values for the current MATLAB session, use the setConfig action. First, set values in an instance of the Simulink.FileGenConfig object. Then, pass the object instance. This example assumes that your system has aNonDefaultCacheFolder and aNonDefaultCodeGenFolder folders.

% Get the current configuration
cfg = Simulink.fileGenControl('getConfig');

% Change the parameters to non-default locations
% for the cache and code generation folders
cfg.CacheFolder = fullfile('C:','aNonDefaultCacheFolder');
cfg.CodeGenFolder = fullfile('C:','aNonDefaultCodeGenFolder');
cfg.CodeGenFolderStructure = 'TargetEnvironmentSubfolder';

Simulink.fileGenControl('setConfig', 'config', cfg);

Set File Generation Control Parameters Directly

You can set file generation control parameter values for the current MATLAB session without creating an instance of the Simulink.FileGenConfig object. This example assumes that your system has aNonDefaultCacheFolder and aNonDefaultCodeGenFolder folders.
myCacheFolder = fullfile('C:','aNonDefaultCacheFolder');
myCodeGenFolder = fullfile('C:','aNonDefaultCodeGenFolder');

Simulink.fileGenControl('set', 'CacheFolder', myCacheFolder, ...
    'CodeGenFolder', myCodeGenFolder, ...
    'CodeGenFolderStructure', ...
    Simulink.filegen.CodeGenFolderStructure.TargetEnvironmentSubfolder);

If you do not want to generate code for different target environments in separate folders,
for 'CodeGenFolderStructure', specify the value

**Reset File Generation Control Parameters**

You can reset the file generation control parameters to values from Simulink preferences.

Simulink.fileGenControl('reset');

**Create Simulation Cache and Code Generation Folders**

To create file generation folders, use the set action with the 'createDir' option. You can keep previous file generation folders on the MATLAB path through the 'keepPreviousPath' option.

% 
myCacheFolder = fullfile('C:','aNonDefaultCacheFolder');
myCodeGenFolder = fullfile('C:','aNonDefaultCodeGenFolder');

Simulink.fileGenControl('set', ...
    'CacheFolder',myCacheFolder, ...
    'CodeGenFolder',myCodeGenFolder, ...
    'keepPreviousPath',true, ...
    'createDir',true);

**Input Arguments**

**Action — Specify action**

'reset' | 'set' | 'setConfig'
Specify an action that uses the file generation control parameters of the current MATLAB session:

- 'reset' – Reset file generation control parameters to values from Simulink preferences.
- 'set' – Set file generation control parameters for the current MATLAB session by directly passing values.
- 'setConfig' – Set file generation control parameters for the current MATLAB session by using an instance of a Simulink.FileGenConfig object.

**Name-Value Pair Arguments**

Specify optional comma-separated pairs of Name,Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside quotes. You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

Example: `Simulink.fileGenControl('Action', 'Name', 'Value');`

**config** — Specify instance of Simulink.FileGenConfig object handle

Specify the Simulink.FileGenConfig object instance containing file generation control parameters that you want to set.

Option for setConfig.

Example: `Simulink.fileGenControl('setConfig', 'config', cfg);`

**CacheFolder** — Specify simulation cache folder character vector

Specify a simulation cache folder path value for the CacheFolder parameter.

Option for set.

Example: `Simulink.fileGenControl('set', 'CacheFolder', 'myCacheFolder');`

**CodeGenFolder** — Specify code generation folder character vector

Specify a code generation folder path value for the CodeGenFolder parameter. You can specify an absolute path or a path relative to build folders. For example:
• 'C:\Work\mymodelsimcache' and '/mywork/mymodelgencode' specify absolute paths.
• 'mymodelsimcache' is a path relative to the current working folder (pwd). The software converts a relative path to a fully qualified path at the time the CacheFolder or CodeGenFolder parameter is set. For example, if pwd is '/mywork', the result is '/mywork/mymodelsimcache'.
• '../test/mymodelgencode' is a path relative to pwd. If pwd is '/mywork', the result is '/test/mymodelgencode'.

Option for set.
Example: Simulink.fileGenControl('set', 'CodeGenFolder', myCodeGenFolder);

CodeGenFolderStructure — Specify generated code folder structure

Specify the layout of subfolders within the generated code folder:
• Simulink.filegen.CodeGenFolderStructure.ModelSpecific (default) – Place generated code in subfolders within a model-specific folder.
• Simulink.filegen.CodeGenFolderStructure.TargetEnvironmentSubfolder – If models are configured for different target environments, place generated code for each model in a separate subfolder. The name of the subfolder corresponds to the target environment.

Option for set.
Example: Simulink.fileGenControl('set', 'CacheFolder', myCacheFolder, ... 'CodeGenFolder', myCodeGenFolder, ... 'CodeGenFolderStructure', ... Simulink.filegen.CodeGenFolderStructure.TargetEnvironmentSubfolder);

keepPreviousPath — Keep previous folder paths on MATLAB path
false (default) | true

Specify whether to keep the previous values of CacheFolder and CodeGenFolder on the MATLAB path:
• true – Keep previous folder path values on MATLAB path.
false (default) - Remove previous older path values from MATLAB path.

Option for reset, set, or setConfig.

Example: Simulink.fileGenControl('reset', 'keepPreviousPath', true);

createDir — Create folders for file generation

false (default) | true

Specify whether to create folders for file generation if the folders do not exist:

• true – Create folders for file generation.
• false (default) – Do not create folders for file generation.

Option for set or setConfig.

Example: Simulink.fileGenControl('set', 'CacheFolder', myCacheFolder, 'CodeGenFolder', myCodeGenFolder, 'keepPreviousPath', true, 'createDir',true);

Avoid Naming Conflicts

Using Simulink.fileGenControl to set CacheFolder and CodeGenFolder adds the specified folders to your MATLAB search path. This function has the same potential for introducing a naming conflict as using addpath to add folders to the search path. For example, a naming conflict occurs if the folder that you specify for CacheFolder or CodeGenFolder contains a model file with the same name as an open model. For more information, see “What Is the MATLAB Search Path?” (MATLAB) and “Files and Folders that MATLAB Accesses” (MATLAB).

To use a nondefault location for the simulation cache folder or code generation folder:

1. Delete any potentially conflicting artifacts that exist in:

   • The current working folder, pwd.
   • The nondefault simulation cache and code generation folders that you intend to use.

2. Specify the nondefault locations for the simulation cache and code generation folders by using Simulink.fileGenControl or Simulink preferences.
Output Arguments

cfg — Current values of file generation control parameters
object handle

Instance of a Simulink.FileGenConfig object, which contains the current values of file generation control parameters.

See Also
“Simulation cache folder” | “Code generation folder” | Code generation folder structure

Topics
“Manage Build Process Folders” (Simulink Coder)
“Share Build Artifacts for Faster Simulation and Code Generation”

Introduced in R2010b
Simulink.findBlocks

Find blocks in Simulink models

Syntax

\[
\begin{align*}
bl &= \text{Simulink.findBlocks}(sys) \\
bl &= \text{Simulink.findBlocks}(sys, \text{options}) \\
bl &= \text{Simulink.findBlocks}(sys, \text{Param1, Value1,}, ..., \text{ParamN, ValueN}) \\
bl &= \text{Simulink.findBlocks}(sys, \text{Param1, Value1,}, ..., \text{ParamN, ValueN, options})
\end{align*}
\]

Description

\[
\begin{align*}
\text{bl} &= \text{Simulink.findBlocks}(sys) \text{ returns handles to all blocks in the model or subsystem } sys. \\
\text{bl} &= \text{Simulink.findBlocks}(sys, \text{options}) \text{ finds blocks that match the criteria specified by a } \text{Simulink.FindOptions} \text{ object.} \\
\text{bl} &= \text{Simulink.findBlocks}(sys, \text{Param1, Value1,}, ..., \text{ParamN, ValueN}) \text{ finds blocks whose block parameters have the specified values.} \\
\text{bl} &= \text{Simulink.findBlocks}(sys, \text{Param1, Value1,}, ..., \text{ParamN, ValueN, options}) \text{ finds blocks whose parameters have the specified values and that match the criteria specified by a } \text{FindOptions} \text{ object.}
\end{align*}
\]

Examples

Find Blocks in a Model

Return handles for all blocks in the model vdp.

```
load_system(vdp);
bl = Simulink.findBlocks('vdp')
```
Return block names.

```matlab
bl = getfullscreen(Simulink.findBlocks('vdp'))
```

```matlab
bl =

12×1 cell array

{vdp/Fcn}  
{vdp/More Info}  
{vdp/More Info/Model Info}  
{vdp/Mu}  
{vdp/Mux}  
{vdp/Product}  
{vdp/Scope}  
{vdp/Sum}  
{vdp/x1}  
{vdp/x2}  
{vdp/Out1}  
{vdp/Out2}

Return block handles for the block whose name is `Mu`.

```matlab
Simulink.findBlocks('vdp','Name','Mu')
```
ans = 8.0001

**Input Arguments**

*sys* — Model or subsystem to find blocks in character vector | string array

Model or subsystem to find blocks in, specified as a character vector or string array. Example: 'vdp' "f14/Aircraft Dynamics Model"

*options* — Search constraints

Simulink.FindOptions object

Search constraints, specified as a Simulink.FindOptions object.

**Output Arguments**

*bl* — Search results

array of handles

Search results, returned as an array of handles.

**See Also**

Simulink.FindOptions | Simulink.allBlockDiagrams | Simulink.findBlocksOfType

**Topics**

“Model Parameters” on page 6-2
“Block-Specific Parameters” on page 6-132

**Introduced in R2018a**
Simulink.findBlocksOfType

Find specified type of block in Simulink models

Syntax

bl = Simulink.findBlocksOfType(sys,type)
bl = Simulink.findBlocksOfType(sys,type,options)
bl = Simulink.findBlocksOfType(sys,
    type,Param1,Value1,...,ParamN,ValueN)
bl = Simulink.findBlocksOfType(sys,
    type,Param1,Value1,...,ParamN,ValueN,options)

Description

bl = Simulink.findBlocksOfType(sys,type) returns handles to all blocks of the
specified type in the model or subsystem sys.

bl = Simulink.findBlocksOfType(sys,type,options) matches the criteria
specified by a FindOptions object.

bl = Simulink.findBlocksOfType(sys,
    type,Param1,Value1,...,ParamN,ValueN) finds blocks whose parameters have the
specified values.

bl = Simulink.findBlocksOfType(sys,
    type,Param1,Value1,...,ParamN,ValueN,options) finds blocks whose
parameters have the specified values and that match the criteria specified by a
FindOptions object.

Examples

Find Blocks of a Type in Model

Find all blocks of type Gain in the model vdp.
load_system('vdp');
Simulink.findBlocksOfType('vdp','Gain')

ans =
     7.0001

To return block names instead of handles, use getfullname.
getfullname(Simulink.findBlocksOfType('vdp','Gain'))

ans =
     'vdp/Mu'

Find Blocks of a Type Using Search Options

Load the model sldemo_clutch. Then, create a FindOptions object and use it to constrain the search of GoTo blocks in the model to the Unlocked system.

load_system('sldemo_clutch');
f = Simulink.FindOptions('SearchDepth',1);
bl = Simulink.findBlocksOfType('sldemo_clutch/Unlocked','Goto',f)

bl =
     166.0001
     167.0001

Input Arguments

sys — Model or subsystem to find blocks in
    character vector | string array

Model or subsystem to find blocks in, specified as a character vector or string array.
Example: 'vdp' "f14/Aircraft Dynamics Model"

type — Block type
    character vector | string array
Block type, specified as a character vector or string array. Use `get_param` with the 'BlockType' parameter to get the block type.

**options — Search constraints**
simulink.FindOptions object

Search constraints, specified as a `Simulink.FindOptions` object.

Example: `Simulink.FindOptions('SearchDepth',1)

### Output Arguments

**bl — Search results**
array of handles

Search results, returned as an array of handles.

### See Also

`Simulink.FindOptions` | `Simulink.allBlockDiagrams` | `Simulink.findBlocks`

### Topics

“Model Parameters” on page 6-2
“Block-Specific Parameters” on page 6-132

**Introduced in R2018a**
Simulink.findIntEnumType

Find enumeration classes defined by Simulink.defineIntEnumType

Syntax

result = Simulink.findIntEnumType(typeName)
result = Simulink.findIntEnumType()

Description

result = Simulink.findIntEnumType(typeName) returns the meta.class object for class type that is defined by Simulink.defineIntEnumType. Use the returned meta.class object to query attributes of the enumeration class. If the class does not exist, the function returns an empty meta.class object.

result = Simulink.findIntEnumType() returns meta.class objects for all enumeration classes that are defined by Simulink.defineIntEnumType. Use the returned meta.class objects to query attributes of the enumeration classes.

Examples

Find a Specific Dynamic Enumerated Data Type

Define an enumeration type.

Simulink.defineIntEnumType('myEnumType', {'e1', 'e2'}, [1 2]);

Check for the enumeration type that you have created.

myResult = Simulink.findIntEnumType('myEnumType')

myResult =

class with properties:
Find All Dynamic Enumerated Data Types

Define two enumeration types.

```matlab
Simulink.defineIntEnumType('myEnumType1', {'e1', 'e2'}, [1 2]);
Simulink.defineIntEnumType('myEnumType2', {'e3', 'e4'}, [3 4]);
```

Check for the enumeration types that you have created.

```matlab
myResult = Simulink.findIntEnumType()
```

### Input Arguments

- **typeName** — Name of enumeration class
  
  character vector or string
  
  Name of a specific enumeration class that is defined by `Simulink.defineIntEnumType`, specified as a character vector or string.
  
  Example: 'myEnumType'
  
  **Data Types:** char | string
Output Arguments

result — Search results
array of meta.class objects

Search result, returned as an array of meta.class objects. If there are no enumeration classes, the array is empty.

See Also

enumeration | Simulink.clearIntEnumType | Simulink.defineIntEnumType

Introduced in R2018b
Simulink.findTemplates

Find model or project templates with specified properties

Syntax

filename = Simulink.findTemplates(templatename)
filename = Simulink.findTemplates(templatename,Name,Value)
[filename,info] = Simulink.findTemplates(templatename)

Description

filename = Simulink.findTemplates(templatename) returns the names and TemplateInfo objects for all matching templates that include templatename.

filename = Simulink.findTemplates(templatename,Name,Value) also specifies additional template properties as one or more Name, Value pair arguments.

[filename,info] = Simulink.findTemplates(templatename) returns the names and TemplateInfo objects for all matching templates.

Examples

Find a Particular Template

Get the full path to the default model template.

filename = Simulink.findTemplates('factory_default_model');

Find All Templates With Specified Folders or Authors

Get all templates inside folders called work.

filename = Simulink.findTemplates('work/')
Get all templates for which the Author property includes the character vector Smith.

```matlab
filename = Simulink.findTemplates('*','Author','Smith')
```

### Find All DSP Templates and Get TemplateInfo Objects

Get the paths to all DSP model templates, and `sltemplate.TemplateInfo` objects for each of them.

```matlab
[filename,info] = Simulink.findTemplates('dsp*','Type','Model');
```

### Input Arguments

**templatename** — Template name

Character vector

Template name, specified as a character vector containing a portion of a file name, which can contain the wildcard asterisk character “*”.

Example:

```
Data Types: char
```

### Name-Value Pair Arguments

Specify optional comma-separated pairs of **Name, Value** arguments. **Name** is the argument name and **Value** is the corresponding value. **Name** must appear inside quotes. You can specify several name and value pair arguments in any order as **Name1, Value1,...,NameN, ValueN**.

You can specify regular expressions for any of the Value character vectors, e.g., including the wildcard asterisk character “*”.

Example: 'Author','*son'

**Type** — Model, library, or project

A character vector.

Model, library, or project template type, specified as a character vector for model, library, or project.
Example: 'Simscape'
Data Types: char

**Title — Title of template**
character vector

Title of template, specified as a character vector.
Example: 'Simscape'
Data Types: char

**Group — Group of template**
character vector

Group of template, specified as a character vector. On the Start Page, templates are shown under group headings.
Example: 'Simscape'
Data Types: char

**Author — Author of template**
character vector

Author of template, specified as a character vector.
Data Types: char

**Description — Description of template**
character vector

Description of template, specified as a character vector.
Data Types: char

**Output Arguments**

**filename — Template name**
character vector | cell array of character vectors

Template names of matching templates, returned as character vectors.
info — Template information

template info objects | array of template info objects

Template information of matching templates, returned as sltemplate.TemplateInfo objects.

See Also
Simulink.createFromTemplate | Simulink.exportToTemplate

Topics
“Create and Open Models”
“Create a Template from a Model”
“Using Templates to Create Standard Project Settings”

Introduced in R2016a
Simulink.findVars

Analyze relationship between variables and blocks in models

Syntax

[variables] = Simulink.findVars(context)
[variables] = Simulink.findVars(context,variablefilter)
[variables] = Simulink.findVars(___ ,Name,Value)

Description

[variables] = Simulink.findVars(context) finds and returns variables that are used in the blocks and models specified by context, including subsystems and referenced models. The function returns an empty vector if context does not use any variables.

[variables] = Simulink.findVars(context,variablefilter) finds only the variables or enumerated types that are specified by variablefilter. For example, use this syntax to determine where a variable is used in a model.

[variables] = Simulink.findVars(___ ,Name,Value) finds variables with additional options specified by one or more Name,Value pair arguments. For example, you can search for unused variables. You can also search for enumerated data types that are used in context, in addition to variables.

Examples

Variables in Use in a Model

Find variables used by MyModel.

variables = Simulink.findVars('MyModel');
**Specific Variable in Use in a Model**

Find all uses of the base workspace variable k by MyModel. Use the cached results to avoid compiling MyModel.

```matlab
variables = Simulink.findVars('MyModel','Name','k',
    'SearchMethod','cached','SourceType','base workspace');
```

**Regular Expression Matching**

Find all uses of a variable whose name matches the regular expression ^trans.

```matlab
variables = Simulink.findVars('MyModel','Regexp','on',
    'Name','^trans');
```

**Variables Common to Two Models**

Given two models, find the variables used by the first model, the second, and both

```matlab
model1Vars = Simulink.findVars('model1');
model2Vars = Simulink.findVars('model2');
commonVars = intersect(model1Vars,model2Vars);
```

**Variables Not Used in a Model**

Find the variables that are defined in the model workspace of MyModel but that are not used by the model.

```matlab
unusedVars = Simulink.findVars('MyModel','FindUsedVars','off',
    'SourceType','model workspace');
```

**Specific Variable Not Used in a Model**

Determine if the base workspace variable k is not used by MyModel.
varObj = Simulink.VariableUsage('k','base workspace');
unusedVar = Simulink.findVars('MyModel',varObj,
'FindUsedVars','off');

**Variables Used by a Block**

Find the variables that are used by the block Gain1 in MyModel.

variables = Simulink.findVars('MyModel',
'Users','MyModel/Gain1');

**Variables Used in a Model Reference Hierarchy**

Find the variables that are used in a model reference hierarchy. Begin the search with the model MyNestedModel, and search the entire hierarchy below MyNestedModel.

variables = Simulink.findVars('MyNestedModel','SearchReferencedModels','on');

**Variables and Enumerated Types Used in a Model**

Find variables and enumerated types that are used in MyModel.

varsAndEnumTypes = Simulink.findVars('MyModel','IncludeEnumTypes','on');

**Input Arguments**

context — Models and blocks to search
character vector | string | cell array of character vectors | string array

Models and blocks to search, specified as a character vector, string, cell array of character vectors, or a string array. You can specify context in one of the following ways:

- The name of a model. For example, ('vdp') specifies the model vdp.slx.
- The name or path of a block or masked block. For example, ('vdp/Gain1') specifies a block named Gain1 at the root level of the model vdp.slx.
• A cell array of model or block names.

Data Types: char | cell

**variablefilter** — Specific variables to find

Array of Simulink.VariableUsage objects

Specific variables to find, specified as an array of Simulink.VariableUsage objects. Each Simulink.VariableUsage object identifies a variable to find.

**Name-Value Pair Arguments**

Specify optional comma-separated pairs of Name, Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside quotes. You can specify several name and value pair arguments in any order as Name1, Value1,...,NameN, ValueN.

Example: 'FindUsedVars','off'

**FindUsedVars** — Find variables that are used or not used

'on' (default) | 'off'

Flag to find variables that are explicitly used or not used, specified as the comma-separated pair consisting of 'FindUsedVars' and 'on' or 'off'. If you specify 'FindUsedVars' as 'off', the function finds variables that are not used in context but that are defined in the workspace specified by SourceType.

Example: 'FindUsedVars','off'

**IncludeEnumTypes** — Find enumerated types that are used

'off' (default) | 'on'

Flag to find enumerated data types that are used, specified as the comma-separated pair consisting of 'IncludeEnumTypes' and 'on' or 'off'. The function finds enumerated types that are used explicitly in context as well as types that define variables that are used in context.

The function does not report details about which blocks or objects are using the enumerated types.

If you specify SourceType as 'base workspace', 'model workspace', or 'mask workspace', the function does not report enumerated types because those sources cannot define enumerated types.
You cannot find unused enumerated types by specifying FindUsedVars as 'off'.
Example: 'IncludeEnumTypes','on'

**RegExp — Enable regular expression matching**

'off' (default) | 'on'

Flag to enable regular expression matching for input arguments, specified as the comma-separated pair consisting of 'RegExp' and 'on'. You can match only input arguments that have character vector values.
Example: 'RegExp','on'

**SearchMethod — Compile status**

'compiled' (default) | 'cached'

Compile status, specified as the comma-separated pair consisting of 'SearchMethod' and one of these values:

- 'compiled' — Return up-to-date results by compiling every model in the search context before search.
- 'cached' — Return quicker results by using results cached during the previous compile.

Example: 'SearchMethod','compiled'

**SearchReferencedModels — Enable search in referenced models**

'off' (default) | 'on'

Flag to enable search in referenced models, specified as the comma-separated pair consisting of 'SearchReferencedModels' and 'on'.

If a referenced model uses a global variable, such as a variable stored in a data dictionary, the function returns this global variable when called on the parent model. If SearchReferencedModels is set to 'on', the Simulink.VariableUsage object returned by the function specifies a block in the referenced model as the variable user. Otherwise, the object specifies the Model block as the user.

Example: 'SearchReferencedModels','on'

**Name — Name of a variable or enumerated type to search for**

character vector | string
Name of a variable or enumerated data type to search for, specified as the comma-separated pair consisting of 'Name' and a character vector or string.

Example: 'Name','trans'

Data Types: char

**SourceType** — **Workspace or source defining the variables or enumerated types**

character vector | string

Workspace or source defining the variables, specified as the comma-separated pair of 'SourceType' and one of these options:

- 'base workspace'
- 'model workspace'
- 'mask workspace'
- 'data dictionary'

The function filters results for variables that are defined in the specified source.

Example: 'SourceType','base workspace'

If you search for enumerated data types by specifying 'IncludeEnumTypes' as 'on', 'SourceType' represents the way an enumerated type is defined. You can specify one of these options:

- 'MATLAB file'
- 'dynamic class'
- 'data dictionary'

The function filters results for enumerated types that are defined in the specified source.

Example: 'SourceType','MATLAB file'

If you do not specify SourceType, the function does not filter results by source.

**Users** — **Name of block to search for variables**

character vector | string

Name of specific block to search for variables, specified as the comma-separated pair consisting of 'Users' and a character vector or string.

To search a set of specific blocks, enable regular expression matching by specifying RegExp as 'on' and use regular expressions in the character vector. For example, you
can specify 'Users','MyModel/Gain* to search all blocks in MyModel whose names begin with Gain.

Example: 'Users','MyModel/Gain1'
Example: 'Users','MyModel/mySubsystem/Gain2'
Example: 'Users','MyModel/Gain*

**Limitations**

Simulink.findVars does not work with these constructs:

- MATLAB code in scripts and initialization and callback functions
- Libraries and blocks in libraries
- Variables in MATLAB Function blocks, except for input arguments

However, Simulink.findVars can find enumerated types anywhere they are used in MATLAB Function blocks.

- Calls directly to MATLAB from the Stateflow action language
- S-functions that use data type variables registered using ssRegisterDataType

To make the variables searchable, use ssRegisterTypeFromNamedObject instead.

- Variables referenced by machine-parented data in Stateflow.
- Variables inside a string.

Simulink.findVars discovers variable usage in inactive subsystem variants only if you select Analyze all choices during update diagram and generate preprocessor conditionals in the Variant Subsystem block dialog box. If you do not select this check box, the function does not discover variable usage in inactive variants.

**See Also**

Simulink.VariableUsage | find_system | intersect

**Topics**

“Search Using Model Explorer”
“Manage Design Data”
Introduced in R2010a
Simulink.fmuexport.ExportSimulinkProjectToFMU

Export project as Functional Mockup Unit (FMU)

Syntax

Simulink.fmuexport.ExportSimulinkProjectToFMU(project,model,Name,Value)

Description

Simulink.fmuexport.ExportSimulinkProjectToFMU(project,model,Name,Value) exports a project as a Functional Mockup Unit (FMU) and sets the model as the main interface model.

Examples

Export Model and Set Logo

Export EngineGasDynamics.slx in the current project:

Simulink.fmuexport.ExportSimulinkProjectToFMU(slproject.getCurrentProject, 'EngineGasDynamics.slx', '-fmuname', 'EngineGasDynamics.fmu', '-fmuicon', 'Simulink_Logo.png')

Input Arguments

- **project** — Project to export
  - string (default)
  - Data Types: string

- **model** — Model that defines the main interface
  - string (default)
Data Types: string

**Name-Value Pair Arguments**

Specify optional comma-separated pairs of Name, Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside quotes. You can specify several name and value pair arguments in any order as Name1, Value1, ..., NameN, ValueN.

Example:

- **-fmuName** — The location and name of the generated FMU relative to the current working folder
  string

If not specified, FMU is saved to current directory folder with the same name as the main model.

Data Types: string

- **-description** — The description of the generated FMU
  string (default)

The information is saved in modelDescription.xml inside FMU.

Data Types: string

- **-author** — The author of the generated FMU
  string (default)

The information is saved in modelDescription.xml inside FMU.

Data Types: string

- **-copyright** — The copyright of the generated FMU
  string (default)

The information is saved in modelDescription.xml inside FMU.

Data Types: string

- **-license** — The license of the generated FMU
  string (default)
The information is saved in `modelDescription.xml` inside FMU.

Data Types: `string`

`-fmuicon` — An icon file for the generated FMU

string (default)

The icon file must be in PNG format. If not specified, generated FMU has no icon file.

Data Types: `string`

See Also

Functions

`slproject.create`

Topics

“Export a Model as a Tool-Coupling FMU”

Introduced in R2018b
Simulink.getFileChecksum

Checksum of file

Syntax

checksum = Simulink.getFileChecksum(filename)

Description

checksum = Simulink.getFileChecksum(filename) returns the checksum of the specified file, using the MD5 checksum algorithm. Use the checksum to see if the file has changed compared to a previous checksum. You can use checksums as part of an audit trail.

Use Simulink.getFileChecksum to get a checksum for any file. If the file contents do not change from one checksum to the next, the checksum from Simulink.getFileChecksum stays the same. Otherwise, the checksum is different with each change to the file contents.

For functional information on a model, use Simulink.BlockDiagram.getChecksum instead. Simulink.BlockDiagram.getChecksum looks at the functional aspect of the model. If the functional aspect doesn't change, then Simulink.BlockDiagram.getChecksum returns the same checksum.

For example, if you moved a block, the file contents are different (measured by Simulink.getFileChecksum) but the function of the model is unchanged (measured by Simulink.BlockDiagram.getChecksum).

Examples

Get Checksum of a File

Use fullfile to specify a full path to a file and get the checksum.
filechecksum = Simulink.getFileChecksum(fullfile(matlabroot,'toolbox',...
'matlab','demos','gatlin.mat'));

**Input Arguments**

filename — File name to get checksum for
file of any type

File name to get checksum for, with file extension and optional full path. Use fullfile to specify a full path to a file, or use the form 'C:\Work\filename.mat'.

Example: 'lengthofline.m'

Data Types: char

**Output Arguments**

checksum — Checksum value
character vector

Checksum value in a 32-character vector.

**See Also**

Simulink.BlockDiagram.getChecksum | Simulink.SubSystem.getChecksum

Introduced in R2014b
Simulink.getSuppressedDiagnostics

Return Simulink.SuppressedDiagnostic objects associated with a block, subsystem, or model

Syntax

suppressed_diagnostics = Simulink.getSuppressedDiagnostics(source)

Description

suppressed_diagnostics = Simulink.getSuppressedDiagnostics(source)
returns an array of Simulink.SuppressedDiagnostic objects that are associated with the specified source.

Examples

Get All Simulink.SuppressedDiagnostic Objects on a Specific Block

Using the model from “Suppress Diagnostic Messages Programmatically”, get all suppressed diagnostics associated with a specified block.

Use the Simulink.suppressDiagnostic function to suppress the parameter precision loss warning thrown by the Constant block, one.

Simulink.suppressDiagnostic('Suppressor_CLI_Demo/one','SimulinkFixedPoint:util:fxpParameterPrecisionLoss');

Get the Simulink.SuppressedDiagnostic objects associated with the block.

suppressed_diagnostic = Simulink.getSuppressedDiagnostics('Suppressor_CLI_Demo/one')
suppressed_diagnostic =

SuppressedDiagnostic with properties:
Input Arguments

source — System, block, or model object throwing warning
model | subsystem | block path | block handle

The source of the diagnostic, specified as a model, subsystem, block path, block handle, cell array of block paths, or cell array of block handles.

To get the block path, use the gcb function.

To get the block handle, use the getSimulinkBlockHandle function.

Data Types: char | cell

Output Arguments

suppressed_diagnostics — Suppressed diagnostics
array

Suppressed diagnostics, returned as an array of Simulink.SuppressedDiagnostic objects.

See Also

Simulink.SuppressedDiagnostic | Simulink.SuppressedDiagnostic.restore | Simulink.getSuppressedDiagnostics | Simulink.restoreDiagnostic | Simulink.suppressDiagnostic

Topics

“Suppress Diagnostic Messages Programmatically”
Introduced in R2016b
Simulink.ID.getHandle

Get handle of block, annotation, or Stateflow object

Syntax

hdl = Simulink.ID.getHandle(simulinkID)

Description

hdl = Simulink.ID.getHandle(simulinkID) returns the handle of the block, annotation, or Stateflow object that has the Simulink Identifier simulinkID.

Examples

Get Handle of Model Block

Open the model vdp.

vdp

Get the SID of the Sum block.

Simulink.ID.getSID('vdp/Sum')

ans =
'vedp:8'

Get the handle of the Sum block using the SID.
hdl = Simulink.ID.getHandle('vdp:8')

**Input Arguments**

**simulinkID** — Simulink Identifier
character vector | string scalar

Simulink Identifier of a block, annotation, or Stateflow object, specified as a character vector or string scalar.

Example: 'vdp:4'

**Output Arguments**

**hdl** — handle
double | object handle

Handle of a block, annotation, or Stateflow object, returned as a double or a Stateflow object handle.

**See Also**

Simulink.ID.getSID

**Topics**

“Get a Simulink Identifier”

**Introduced in R2009b**
Simulink.ID.getSID

Get Simulink Identifier of block, annotation, or Stateflow object

Syntax

simulinkID = Simulink.ID.getSID(obj)

Description

simulinkID = Simulink.ID.getSID(obj) returns the Simulink Identifier (SID) of the block, annotation, or Stateflow object obj.

A Simulink Identifier (SID) is a unique and unmodifiable identifier. Use the SID to identify an object throughout its lifetime, even if the name of the object changes.

Examples

Get SID of Model Block

Open the model vdp.

vdp

Get the SID of the Mu block.

simulinkID = Simulink.ID.getSID('vdp/Mu')

Get SID of Annotation

Open the model vdp.

vdp
Select the annotation at the top of the diagram. To get the annotation object, use the function `getCurrentAnnotation`.

```matlab
ann = getCurrentAnnotation
```

Get the SID of the annotation.

```matlab
simulinkID = Simulink.ID.getSID(ann)
```

**Get SID of Model Block Using Handle**

Open the model `vdp`.

```matlab
vdp
```

Get the handle of the Product block.

```matlab
hdl = getSimulinkBlockHandle('vdp/Product')
```

Get the SID of the Product block using the handle as the input argument.

```matlab
simulinkID = Simulink.ID.getSID(hdl)
```

**Input Arguments**

`obj` — Name or handle of block, annotation, or Stateflow object

character vector | string scalar | handle

Name or handle of a block, annotation, or Stateflow object, specified as a handle, character vector, or string scalar.

Example: `'vdp/Mu'`

**Output Arguments**

`simulinkID` — Simulink Identifier

string scalar

Simulink Identifier (SID), returned as a string scalar.
See Also
Simulink.ID.getHandle | Simulink.ID.hilite

Topics
“Get a Simulink Identifier”

Introduced in R2009b
**Simulink.ID.hilite**

Highlight block, annotation, or Stateflow object

**Syntax**

Simulink.ID.hilite(simulinkID)
Simulink.ID.hilite(simulinkID,style)

**Description**

Simulink.ID.hilite(simulinkID) highlights the block, annotation, or Stateflow object that has the Simulink Identifier simulinkID.

Simulink.ID.hilite(simulinkID,style) uses the specified highlighting style.

**Examples**

**Highlight Block**

Open the model vdp.

dp

Get the Simulink Identifier of the Mu block.

muSID = Simulink.ID.getSID('vdp/Mu')

Highlight the Mu block.

Simulink.ID.hilite(muSID)
Highlight Block Using a Highlight Style

Open the model vdp.

vdp

Get the Simulink Identifier of the Mu block.

muSID = Simulink.ID.getSID('vdp/Mu')

Highlight the Mu block using the style 'fade'.

Simulink.ID.hilite(muSID, 'fade')

Input Arguments

simulinkID — Simulink Identifier
character vector | string scalar

Simulink Identifier of a block, annotation, or Stateflow object, specified as a character vector or string scalar.

Example: 'vdp:4'

style — Highlighting style
'default' (default) | character vector | string scalar

Highlighting style, specified as one of these values. You can customize the appearance of any of the styles. See “Customize a Highlighting Style” on page 2-393.

- 'default' — Default color scheme: red outline, yellow fill.
- 'none' — Clears the highlight.

To clear all highlighting, in the Simulink Editor, select Display > Remove Highlighting.

- 'debug' — Uses default color scheme.
- 'different' — Applies red outline, white fill.
- 'error' — Uses default color scheme.
- 'fade' — Applies gray outline, white fill.
• 'find' — Applies dark blue outline, blue fill.
• 'lineTrace' — Applies red outline, blue fill.
• 'unique' — Dark blue outline, white fill.
• 'user1','user2','user3','user4','user5' — Applies custom highlight: black outline, white fill by default (that is, no highlight).

In addition, you can use these color schemes. The first word is the outline and the second is the fill color:

• 'orangeWhite'
• 'blackWhite'
• 'redWhite'
• 'blueWhite'
• 'greenWhite'

See Also
Simulink.ID.getSID

Topics
“Get a Simulink Identifier”

Introduced in R2009b
Simulink.importExternalCTypes

Generate Simulink representations of custom data types defined by C or C++ code

Syntax

importInfo = Simulink.importExternalCTypes(headerFiles)
importInfo = Simulink.importExternalCTypes(modelName)
importInfo = Simulink.importExternalCTypes(____,Name,Value)

Description

importInfo = Simulink.importExternalCTypes(headerFiles) parses the C or C++ header files (.h or .hpp) identified by headerFiles for typedef, struct, and enum type definitions, and generates Simulink representations of the types. The output, importInfo, identifies the successfully and unsuccessfully imported types.

You can use the Simulink representations to:

- Reuse your existing algorithmic C code and, through simulation, test its interaction with your Simulink control algorithm. For an example that shows how to use the Legacy Code Tool, see “Integrate C Function Whose Arguments Are Pointers to Structures”.
- Generate code (Simulink Coder) that reuses the types and data that your existing code defines. You can then integrate and compile the generated and existing code into a single application. For an example, see “Exchange Structured and Enumerated Data Between Generated and External Code” (Embedded Coder).
- Create and organize data (signals, parameters, and states) in a model by using standard data types that your organization defines in C code.
  - To create structures of signals in Simulink, use nonvirtual buses. See “Getting Started with Buses”.
  - To create structures of parameters, use MATLAB structures and Simulink.Parameter objects. See “Organize Related Block Parameter Definitions in Structures”.

2-759
• To create enumerated data, see “Use Enumerated Data in Simulink Models”.
• To match a primitive typedef statement, use a Simulink.AliasType object to set parameter and signal data types in a model.

By default, the function:

• Imports an enumerated type by generating a script file that derives an enumeration class from Simulink.IntEnumType, as described in “Define Simulink Enumerations”. If necessary, you can then edit the class definition to customize it (for example, by implementing the addClassNameToEnumNames method).
• Imports a structure type by generating a Simulink.Bus object in the base workspace.
• Imports a primitive typedef statement by generating a Simulink.AliasType object in the base workspace.
• Interprets generic C data types, such as int or short, according to the word lengths of your host computer. For example, for most modern machines, int has a 32-bit word length, so the function represents an int structure field as a bus element that uses the Simulink data type int32.

To override this default behavior, identify your target hardware board by using the HardwareImplementation pair argument.

For additional information about default behavior, see “Tips” on page 2-778.

importInfo = Simulink.importExternalCTypes(modelName) generates Simulink representations of custom C data types by analyzing a model that you identify with modelName. When you use the Simulation Target configuration parameters in a model to identify header files for inclusion during simulation, use this syntax to import types for the purpose of simulating the model on your host computer. The function interprets generic C data types according to the word lengths of your host computer.

When you use this syntax, do not use pair arguments, such as HardwareImplementation, that can conflict with the configuration parameters of the target model. When you use such pair arguments with this syntax, the function generates a warning.

importInfo = Simulink.importExternalCTypes(____,Name,Value) specifies additional options using one or more name-value pair arguments. You can use this syntax to:
• Specify the names of types to import by using the Names pair argument.
• Control the way that Simulink stores the imported types, for example, by generating the types in a Simulink data dictionary. Use the MATFile and DataDictionary pair arguments.
• Control the way that the function interprets generic C data types. Use the HardwareImplementation pair argument.
• Maintain synchrony between the C-code definitions and the Simulink representations by attempting to import the updated C-code definitions again. You can choose whether to overwrite the existing Simulink representations. Use the Overwrite and Verbose pair arguments.

Examples

Import Simple Structure and Enumerated Types

This example shows how to generate Simulink representations of a C structure type (struct) and an enumerated (enum) data type from a header file.

1 In your current folder, create the file ex_cc_simpleTypes.h.

    typedef enum {
        PWR_LOSS = 0,                 /* Default value */
        OVERSPD,
        PRESS_LOW,
    } fault_T;

    typedef struct {
        double coeff;
        double init;
    } params_T;

2 Generate Simulink representations of the types by calling Simulink.importExternalCTypes.

    Simulink.importExternalCTypes('ex_cc_simpleTypes.h');

    The function creates a Simulink.Bus object, params_T, in the base workspace.

3 To inspect the properties of the object, open the Bus Editor.

    buseditor
Each bus element uses a name and a data type (double) that match the corresponding structure field in ex_cc_simpleTypes.h.

4 In your current folder, inspect the generated file, fault_T.m, which defines the enumerated type fault_T as an enumeration class.

You can use the bus object and the enumeration class to set signal and parameter data types in Simulink models.

Import Structure Type Whose Fields Use Custom Data Types

This example shows how to generate a Simulink representation of a structure type whose fields use custom data types (typedef).

Create the file ex_integer_aliases.h in your current folder.

typedef int sint_32;
typedef unsigned short uint_16;

Create the file ex_cc_struct_alias.h in your current folder.

#include "ex_integer_aliases.h"

typedef struct {
    sint_32 accum;
    uint_16 index;
} my_ints_T;

Import the structure type into Simulink as a Simulink.Bus object in the base workspace. Import the typedef statements as Simulink.AliasType objects.

Simulink.importExternalCTypes('ex_cc_struct_alias.h');

Inspect the data types of the bus elements in the bus object. For example, inspect the DataType property of the first bus element, which corresponds to the structure field accum.

my_ints_T.Elements(1)

ans =
BusElement with properties:

Name: 'accum'
Complexity: 'real'
Dimensions: 1
DataType: 'sint_32'
Min: []
Max: []
DimensionsMode: 'Fixed'
SampleTime: -1
Unit: ''
Description: ''

The `Simulink.importExternalCTypes` function uses the generated `Simulink.AliasType` objects to set the data types of the bus elements.

Inspect the `Simulink.AliasType` objects in the base workspace. For example, the object named `sint_32` corresponds to one of the `typedef` statements in `ex_integer_aliases.h`.

`sint_32`

`sint_32` =

AliasType with properties:

Description: ''
DataScope: 'Imported'
HeaderFile: 'ex_integer_aliases.h'
BaseType: 'int32'

For most host computers (which the function targets by default), the word length of `int` is 32 bits and the word length of `unsigned short` is 16 bits. The function maps `int` and `unsigned short` to the Simulink types `int32` and `uint16`.

If you have Embedded Coder, the code that you generate from the model can use `sint_32` and `uint_16` instead of the standard data type names, `int32_T` and `uint16_T`. 

2-763
Store Imported Types in Data Dictionary

This example shows how to store the imported data types in a Simulink data dictionary. A data dictionary stores data specifications (such as for signals and block parameter values), data types, and other design data for one or more Simulink models.

In your current folder, create the file ex_cc_simpleTypes.h.

```c
typedef enum {
    PWR_LOSS = 0,            /* Default value */
    OVERSPD,
    PRESS_LOW,
} fault_T;

typedef struct {
    double coeff;
    double init;
} params_T;
```

Create a subfolder called myDictionaries.

```bash
mkdir('myDictionaries')
```

Generate Simulink representations of the types by calling Simulink.importExternalCTypes. Permanently store the type definitions by creating a new data dictionary, ex_cc_myTypes.sldd, in the new subfolder.

```bash
Simulink.importExternalCTypes('ex_cc_simpleTypes.h',
    'DataDictionary','ex_cc_myTypes.sldd',
    'OutputDir','myDictionaries');
```

To inspect the contents of the dictionary, set your current folder to myDictionaries and double-click the dictionary file.

To use the Simulink representations in the dictionary, you must link a model or models to the dictionary. See “Migrate Models to Use Simulink Data Dictionary”.

Import Only Specified Types

This example shows how to generate Simulink representations only for enumerated and structure data types that you identify by name.
In your current folder, create the file `ex_cc_manySimpleTypes.h`. The file defines three structure types: `params_T`, `signals_T`, and `states_T`.

```c
typedef struct {
    double coeff;
    double init;
} params_T;

typedef struct {
    double flow_rate;
    double steam_press;
} signals_T;

typedef struct {
    double accum;
    double error;
} states_T;
```

Generate Simulink representations only for `params_T` and `signals_T`.

```matlab
Simulink.importExternalCTypes('ex_cc_manySimpleTypes.h',
    'Names',{'params_T','signals_T'});
```


**Import Types for 16-Bit Hardware**

By default, `Simulink.importExternalCTypes` represents an enumerated data type by creating an enumeration class that derives from the built-in class `Simulink.IntEnumType`. When you simulate or generate code from a model that uses the generated class, configuration parameters that you select for the model (for example, on the **Hardware Implementation** pane) determine the specific integer length that `Simulink.IntEnumType` and the enumeration class employ.

By default, the function interprets generic, primitive C data types, such as `short` and `int`, according to the word lengths of your host computer. For example, to represent an `int` structure field, the function typically applies the 32-bit data type `int32` to the corresponding bus element. When you want to simulate and generate code for hardware other than your host computer, use the `HardwareImplementation` pair argument to identify the target hardware and, by extension, the word lengths of the hardware.
This example shows how to import data types from code that you intend to use on 16-bit hardware. For this board, int has a 16-bit length, and each item of enumerated data (enum) consumes 16 bits.

In your current folder, create the file ex_cc_intTypes.h.

typedef enum {
    PWR_LOSS = 0,            /* Default value */
    OVERSPD,
    PRESS_LOW,
} fault_T;

typedef struct {
    int coeff;
    int init;
} params_T;

The code defines an enumerated data type and a structure type whose fields use the generic C data type int.

To generate an accurate Simulink representation of the structure type, first open an existing model or create a new model. For this example, create a new model named ex_hdwImpl_16bit.

In the new model, set Configuration Parameters > Hardware Implementation > Device vendor to Atmel. Set Device type to AVR.

Alternatively, at the command prompt, use these commands to create and configure the model:

```matlab
new_system('ex_hdwImpl_16bit','Model');
set_param('ex_hdwImpl_16bit','ProdHWDeviceType','Atmel->AVR')
```

Generate Simulink representations of the types. To specify the word lengths of the target 16-bit hardware, extract the model configuration parameters (which include the Hardware Implementation settings) as a Simulink.ConfigSet object.

```matlab
configSet = getActiveConfigSet('ex_hdwImpl_16bit');
Simulink.importExternalCTypes('ex_cc_intTypes.h','HardwareImplementation',configSet);
```

The Simulink.Bus object params_T appears in the base workspace. The bus elements, such as coeff, use the Simulink data type int16.
params_T.Elements(1)

ans =

    BusElement with properties:
        Name: 'coeff'
        Complexity: 'real'
        Dimensions: 1
        DataType: 'int16'
        Min: []
        Max: []
        DimensionsMode: 'Fixed'
        SampleTime: -1
        Unit: ''
        Description: ''

In your current folder, the file fault_T.m defines the enumeration class fault_T. The class derives from Simulink.IntEnumType, so you must use model configuration parameters to identify the target hardware and, by extension, the correct native integer length.

**Import Structure Type Whose Fields Use 16-Bit Fixed-Point Data Types**

Create the file ex_cc_fixpt_struct.h in your current folder.

typedef struct {

    int coeff;   /* Word length 16,
                  binary fraction length 7 */

    int init;   /* Word length 16,
                  binary fraction length 3 */

} params_T;

The file defines a structure type whose fields use fixed-point data types. For example, the structure stores the field coeff in a signed, 16-bit integer data type. A binary fraction length of 7 relates the stored integer value to the real-world value.
Suppose that this code operates on 16-bit hardware (such that the generic C data type `int` has a 16-bit word length). To generate a Simulink representation of the type, first create a `coder.HardwareImplementation` object that identifies the hardware.

```matlab
hdw = coder.HardwareImplementation;
hdw.ProdHWDeviceType = 'Atmel->AVR';
```

Generate a Simulink representation of the structure type.

```matlab
Simulink.importExternalCTypes('ex_cc_fixpt_struct.h', ...
    'HardwareImplementation', hdw);
```

The `Simulink.Bus` object, `params_T`, appears in the base workspace. Each bus element, such as `coeff`, uses the data type `int16`.

```matlab
params_T.Elements(1)
```

```matlab
ans =

    BusElement with properties:

    Name: 'coeff'
    Complexity: 'real'
    Dimensions: 1
    DataType: 'int16'
    Min: []
    Max: []
    DimensionsMode: 'Fixed'
    SampleTime: -1
    Unit: ''
    Description: ''
```

`Simulink.importExternalCTypes` cannot infer the fixed-point scaling (binary fraction length) from the C code. You must manually specify the data types of the bus elements. To specify the data types at the command prompt, use the `fixdt` function.

```matlab
params_T.Elements(1).DataType = 'fixdt(1,16,7)';
params_T.Elements(2).DataType = 'fixdt(1,16,3)';
```

To specify the data types interactively (by using the Data Type Assistant), use the Bus Editor.
Manually Synchronize Simulink Representations with C-Code Definitions

This example shows how to maintain the Simulink representations of C data types whose definitions you modify during the life of a modeling project.

Import Custom C Types

Create the file `ex_cc_myTypes_rec.h` in your current folder. The file defines a custom structure type.

```c
typedef struct {
    double flow;
    double pres;
    double tqe;
} sigStructType;
```

Generate a Simulink.Bus object that represents the type.

```c
Simulink.importExternalCTypes('ex_cc_myTypes_rec.h');
```

Modify Type Definition in C Code

In `ex_cc_myTypes_rec.h`, add a field named `spd` to `sigStructType`.

In the same file, create a new structure type, `stateStructType`.

```c
typedef struct {
    double flow;
    double pres;
    double tqe;
    double spd;
} sigStructType;

typedef struct {
    double err;
    double read;
    double write;
} stateStructType;
```
**Attempt to Import Types Again**

Attempt to generate bus objects that represent the types.

```matlab
importInfo = Simulink.importExternalCTypes('ex_cc_myTypes_rec.h');
```

The function generates warnings at the command prompt. Instead of relying on the warnings, you can inspect the output, `importInfo`, to determine whether the function failed to import any types.

```matlab
importInfo.failedToImport.Bus
```

```matlab
ans =
1×1 cell array
{'sigStructType'}
```

The function did not import `sigStructType`. The corresponding bus object in the base workspace still has only three bus elements. To determine the reason that the function did not import `sigStructType`, inspect the `report` field of `importInfo`.

Import `sigStructType` again. This time, overwrite the existing bus object.

```matlab
importInfo = Simulink.importExternalCTypes('ex_cc_myTypes_rec.h',...
    'Names',importInfo.failedToImport.Bus,'Overwrite','on');
```

When you overwrite existing Simulink representations, any customizations that you made to the Simulink representations (such as the application of fixed-point data types to bus elements) are overwritten.

**Input Arguments**

`headerFiles` — Names and paths of header files to parse

character vector | cell array of character vectors | string scalar | string array

Names and paths of header files to parse, specified as a character vector, cell array of character vectors, string, or string array. Include the `.h` or `.hpp` file extension.

If you use a hierarchy of included (#include) header files to define your types, when you specify `HeaderFiles`, you need to identify only the entry-point files. The function parses
the included files as well as the identified entry-point files. If the included files are not in the same folder as the corresponding entry-point file, use the IncludeDirs pair argument to identify the additional folders.

Example: 'myHeader.h'
Example: {'thisHeader.hpp','thatHeader.hpp'}

Data Types: char | cell | string

[modelName — Name of loaded Simulink model for which to import types](#)

cell | string scalar

Name of a loaded Simulink model for which to import types, specified as a character vector or string scalar. A model is loaded if, for example, you open the model or use the load_system function. When you use this argument, the function:

- Searches the model configuration parameters for custom header files and parses those header files for data types to import. Only the configuration parameters on the Simulation Target pane affect this search.

For example, if in the model you set Configuration Parameters > Simulation Target > Insert custom C code in generated > Header file to #include "myTypes.h", the function parses myTypes.h for types to import.

- Interprets generic C data types such as int or short according to the word lengths of your host computer. Do not use the HardwareImplementation pair argument to override this interpretation.

Example: 'myModel'

Data Types: char | string

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name, Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside quotes. You can specify several name and value pair arguments in any order as Name1, Value1, ..., NameN, ValueN.

Example:
Simulink.importExternalCTypes('myHdr.h','DataDictionary','myDictionary.sldd')
**MATFile — Name and path of MAT-file to create for storing generated objects**

'' (empty) (default) | character vector | string scalar

Name and, optionally, path of the MAT-file to create for storing generated Simulink.Bus and Simulink.AliasType objects, specified as a character vector or string. If you do not use MATFile, by default, the function generates the objects in the base workspace.

The function does not generate enumeration definitions in MAT-files.

If you import some struct types and primitive typedef statements by using MATFile and later import some of the same types again by using MATFile, the function entirely replaces the old MAT-file with a new one. The function discards any changes that you made to the contents of the old MAT-file.

You cannot use the MATFile and DataDictionary pair arguments simultaneously.

Example: 'myMat.mat'
Example: 'myMat'
Example: fullfile('subfolder','myMat')

Data Types: char | string

**DataDictionary — Name and path of Simulink data dictionary to use or create for storing types**

'' (default) | character vector | string scalar

Name and, optionally, path of the Simulink data dictionary to use or create for storing generated enumerations and objects, specified as a character vector or string. When you use this pair argument, the function imports enumerated types as Simulink.data.dictionary.EnumTypeDefinition objects, and stores those objects (as well as Simulink.Bus objects and Simulink.AliasType objects) in the target dictionary.

For information about data dictionaries, see “What Is a Data Dictionary?”.  

You can optionally specify a .sldd extension.

You cannot use the DataDictionary and MATFile pair arguments simultaneously.

Example: 'myDict.sldd'
Example: 'myDict'
Example: `fullfile('subfolder','myDict.sldd')`

Data Types: `char` | `string`

**Names — Names of types to import**

' ' (default) | character vector | cell array of character vectors | string scalar | string array

Names of types to import, specified as a character vector, cell array of character vectors, string, or string array. By default, if you do not use `Names`, the function attempts to import all of the custom types that the identified header files define.

To match multiple type names with a single character vector, use an asterisk (*).

Example: 'myEnumType'
Example: {'myEnumType','myStructType'}
Example: 'my*Type'

Data Types: `char` | `cell` | `string`

**Defines — Compiler options to define macros that influence type definitions**

' ' (default) | character vector | string scalar

Compiler options to define macros that influence C type definitions, specified as a character vector, or string scalar. For example, a macro influences a type definition if you enclose the definition in an `#ifdef` block that checks whether the macro is defined.

Use `Defines` to specify macro definitions that you otherwise define through compiler options such as `-D`.

Example: 'SIGSTRUCT=1'
Example: 'SIGSTRUCT=1 ENUM=1'

Data Types: `char` | `string`

**UnDefines — Compiler options to delete macros that influence type definitions**

' ' (default) | character vector | string scalar

Compiler options to delete macros that influence C type definitions, specified as a character vector or string scalar. For example, a macro influences a type definition if you enclose the definition in an `#ifdef` block that checks whether the macro is defined.
Use **UnDefines** to specify macro deletions that you otherwise define through compiler options such as `-U`.

Example: 'SIGSTRUCT'
Example: 'SIGSTRUCT_ENUM'

**Data Types:** char | string

**IncludeDirs — Folders that contain subordinate, included header files**

'' (default) | character vector | cell array of character vectors | string scalar | string array

Folders that contain subordinate, included (#include) header files, specified as a character vector, cell array of character vectors, string, or string array. Use this pair argument to enable the function to locate and parse additional header files on which the primary header files (which you specify with the **headerFiles** argument) depend.

If you use the **modelName** syntax instead of the **headerFiles** syntax, in the target model, you can use the **Simulation Target** configuration parameters to specify include paths. In that case, you do not need to use the **IncludeDirs** pair argument.

Example: 'myHeaders'
Example: fullfile('myProject','myHeaders')
Example:
{fullfile('myProject','myHeaders'),fullfile('myProject','myOtherHeaders')}

**Data Types:** char | cell | string

**OutputDir — Folder for storing generated files**

'' (default) | character vector | string scalar

Folder for storing generated files, specified as a character vector or string. The function places generated files, such as **classdef** script files and data dictionary files, in this folder.

The folder that you specify must exist before you use the function.

Example: 'myDictionaries'
Example: fullfile('myProject','myDictionaries')

**Data Types:** char | string
HardwareImplementation — Word lengths for interpreting generic, primitive C
data types
'' (default) | Simulink ConfigSet object | coder HardwareImplementation object

Word lengths for interpreting generic, primitive C data types, specified as a Simulink ConfigSet or coder HardwareImplementation object.

- To use a Simulink ConfigSet object, you can extract a configuration set from a model by using functions such as getConfigSet and getActiveConfigSet. This technique enables you to use the Configuration Parameters dialog box to identify your target hardware (through the Hardware Implementation configuration parameters).
- To use a coder HardwareImplementation object (which you create and configure programmatically), specify properties of the object, such as ProdHWDeviceType, to identify your target hardware. The object then sets other properties, such as ProdBitPerInt, that reflect the native integer size of the hardware.

The function inspects the object to determine which Simulink integer data types to employ when interpreting generic C data types such as int. For example, if you create a coder HardwareImplementation object to identify 16-bit hardware and then use the function to import a structure type whose fields use the C data type int, the function generates a bus object whose bus elements use the Simulink data type int16. The function uses the production hardware settings, not the test hardware settings.

For more information about hardware implementation settings for Simulink models, see “Configure Run-Time Environment Options” (Simulink Coder).

Overwrite — Specification to overwrite existing Simulink representations
'off' (default) | 'on'

Specification to overwrite existing Simulink representations, specified as 'on' or 'off'. If an imported type already has a representation in Simulink:

- If you specify 'off' or if you do not specify Overwrite, the function does not import the type. In the output argument, importInfo, the failedToImport field identifies the type.
- If you specify 'on', the function overwrites the existing Simulink representation.

If you use the function to import some types into the base workspace or a data dictionary and later customize the generated Simulink representations, when you use the function again and set Overwrite to 'on', the function does not preserve your customizations. These customizations can include:
• In an enumeration class definition, implementing extra methods or modifying the generated methods such as `getDataScope` (see “Customize Simulink Enumeration”).
• Modifying the properties of a generated `Simulink.Bus` or `Simulink.AliasType` object (for example, manually setting the data types of bus elements to a fixed-point data type).

**Verbose — Specification to generate messages for successful import operations**

`'off'` (default) | `'on'`

Specification to generate messages for successful import operations, specified as `'on'` or `'off'`.

• If you specify `'off'` or if you do not specify **Verbose**, the function imports types silently. Messages do not appear in the Command Window unless the function cannot import a type.
• If you specify `'on'`, the function generates a message in the Command Window for each operation during the import process.

**Output Arguments**

**importInfo — Information about types that were imported and not imported**

structure

Information about types that were imported and not imported, returned as a structure with these fields.

**report — Descriptions of types that were imported and not imported**

character vector

Descriptions of types that were imported and not imported, returned as a character vector. Inspect the value of this field to determine the reason that the function could not import a type.

**failedToImport — Types that were not imported**

structure

Types that were not imported, returned as a structure with these fields.
<table>
<thead>
<tr>
<th>Field Name</th>
<th>Field Value</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bus</td>
<td>Cell array of character vectors</td>
<td>Names of structure (struct) types that were not imported.</td>
</tr>
<tr>
<td>Enum</td>
<td>Cell array of character vectors</td>
<td>Names of enumerated types (enum) that were not imported.</td>
</tr>
<tr>
<td>AliasType</td>
<td>Cell array of character vectors</td>
<td>Names of primitive typedef statements that were not imported.</td>
</tr>
</tbody>
</table>

**importedTypes — Types that were successfully imported**

Types that were successfully imported, returned as a structure with these fields.

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Field Value</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bus</td>
<td>Cell array of character vectors</td>
<td>Names of structure (struct) types that were imported. The generated Simulink.Bus objects use these names.</td>
</tr>
<tr>
<td>Enum</td>
<td>Cell array of character vectors</td>
<td>Names of enumerated types (enum) that were imported. The generated enumeration classes or Simulink.data.dictionary.EnumTypeDefinitio n objects use these names.</td>
</tr>
<tr>
<td>AliasType</td>
<td>Cell array of character vectors</td>
<td>Names of primitive typedef statements that were imported. The generated Simulink.AliasType objects use these names.</td>
</tr>
</tbody>
</table>
Limitations

- The function does not support:
  - C data types that do not correspond to a type that Simulink supports. For example, Simulink does not recognize an equivalent for `long double`. For information about data types that Simulink supports, see “Data Types Supported by Simulink”.
  - Pointer types, such as a structure that defines a field whose value is a pointer or a `typedef` statement whose base type is a pointer type.
  - Structures that define a field whose value has more than one dimension. If a field value is a 1-D array, the function creates a bus element that represents a vector, not a matrix.
  - Unions.
  - If a structure field represents fixed-point data, or if a `typedef` statement maps to a fixed-point base type, the function sets the data type of the corresponding bus element or `Simulink.AliasType` object to the relevant Simulink integer type (such as `int16`). The importer cannot determine the fixed-point scaling by parsing the C code. After using the function, you must manually specify the data type of the bus element or the base type of the `Simulink.AliasType` object by using the `fixdt` function.

Tips

- If a MATLAB Function block or Stateflow chart in your model uses an imported enumeration or structure type, configure the model configuration parameters to include (`#include`) the type definition from your external header file. See “Control Imported Bus and Enumeration Type Definitions” (for a MATLAB Function block) and “Access Custom Code Variables and Functions in Stateflow Charts” (Stateflow) and “Integrate Custom Structures in Stateflow Charts” (Stateflow) (for a chart).
  - By default:
    - For an imported enumeration, because the Simulink enumeration class derives from `Simulink.IntEnumType`, when you simulate or generate code from a model, the enumeration uses the integer size that is native to your target hardware. You specify the characteristics of your target hardware by using model configuration parameters such as `Production device vendor and type` and `Native word size in production hardware`. 
• For an imported structure type:
  • The function imports a structure field as numerically complex only if the field
    uses one of the corresponding Simulink Coder structure types as the data type.
    For example, if a structure field in your external code uses the data type
    cint8_T, the function imports the field as a bus element
    (Simulink.BusElement object) whose data type is int8 and whose
    Complexity property is set to 'complex'.
  • For nested structures, the function generates a bus object for each unique
    structure type.
  • For an imported structure type or enumeration, if your external code uses a
    typedef statement to name the type, the name of the generated bus object or
    Simulink enumeration class matches the typedef name. If your code does not use
    a typedef statement, the name of the object or class is struct_type
    or enum_type where type is the tag name of the type. If you do not specify a tag
    name or apply a typedef name, Simulink generates an arbitrary name for the
    object or class.
  • The function configures the generated Simulink representations as imported for
    the purposes of simulation and code generation. For example, for bus objects, the
    function sets the DataScope property to 'Imported' and the HeaderFile
    property to the name of your external header file. To simulate or generate code
    from a model that uses one of these Simulink representations, you must make your
    header file available to the model.
• When you specify files for Simulink.importExternalCTypes to use or generate,
  for example, by using the DataDictionary pair argument:
  • If the existing files to use are in your current folder or on the MATLAB path, you do
    not need to specify a file path. You can specify the file name by itself.
  • To control the folder location of generated files, you can specify paths as well as file
    names. You can also use the OutputDir pair argument.

See Also
Simulink.AliasType | Simulink.Bus | enumeration

Topics
“Data Types Supported by Simulink”
“Data Types for Bus Signals”
“Use Enumerated Data in Simulink Models”
“Control Data Type Names in Generated Code” (Embedded Coder)
“Control Signal Data Types”
“Exchange Data Between External C/C++ Code and Simulink Model or Generated Code” (Simulink Coder)

**Introduced in R2017a**
Simulink.ModelDataLogs.convertToDataset

Convert logging data from Simulink.ModelDataLogs format to Simulink.SimulationData.Dataset format

Syntax

```matlab
convertedDataset = sourceModelDataLogsObject.convertToDataset(convertedDatasetName)
```

Description

**Note** The ModelDataLogs class is supported for backwards compatibility. Starting in R2016a, you cannot log data in the ModelDataLogs format. Signal logging uses the Dataset format. In R2016a or later, when you open a model from an earlier release that had used ModelDataLogs format, the model simulated in use Dataset format.

You can convert signal logging data from ModelDataLogs to Dataset format. Converting to Dataset format makes it easier to post-process with other logged data (for example, logged states), which can also use Dataset format. For more information, see “Convert Logged Data to Dataset Format”.

If you have legacy code that uses the ModelDataLogs API, you can encounter situations that require updates to your code or model. See “Migrate Scripts That Use Legacy ModelDataLogs API”.

```matlab
convertedDataset = sourceModelDataLogsObject.convertToDataset(convertedDatasetName)
```

converts the `sourceModelDataLogsObject` to a Simulink.SimulationData.Dataset object. The name of the converted object is based on `convertedDatasetName`.

The resulting Simulink.SimulationData.Dataset object is a flat list. This list has one element for each Simulink.Timeseries or Simulink.TsArray object in the Simulink.ModelDataLogs object.
Limitations

<table>
<thead>
<tr>
<th>Source of Simulink.ModelDataLogs Logged Data</th>
<th>Conversion Limitation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Referenced model</td>
<td>Loads all ancestors of the referenced model not previously loaded. If any ancestor model does not appear on the MATLAB path, the conversion fails. If the model has changed, or the model ancestors have changed, after Simulink logged the data, the conversion can fail. For example, adding, deleting, or renaming a block after logging can cause conversion failure.</td>
</tr>
<tr>
<td>Variant model or subsystem</td>
<td>The current active variant must be the same one that was active when Simulink logged the data. Otherwise, the conversion fails.</td>
</tr>
<tr>
<td>Frame signal</td>
<td>The conversion fails.</td>
</tr>
<tr>
<td>Mux block</td>
<td>The conversion produces a different Simulink.SimulationData.Dataset object as the dataset than Simulink creates when you simulate the model using the Dataset format for the logged data.</td>
</tr>
<tr>
<td>Stateflow chart</td>
<td>Not supported.</td>
</tr>
</tbody>
</table>

Input Arguments

**sourceModelDataLogsObject**

A Simulink.ModelDataLogs object that you want to convert to a Simulink.SimulationData.Dataset object.

**convertedDatasetName**

Name of the dataset that the conversion process creates.
Output Arguments

convertedDataset

The Simulink.SimulationDataset object that the Simulink.ModelDataLogs.convertToDataset function creates.

For details about the converted dataset, see Simulink.SimulationData.Dataset.

Example

In releases before R2016a, you could log signals using ModelDataLogs format. If you have a MAT-file with signal logging data that uses the ModelDataLogs format, here is how you can convert that data to Dataset format. This example assumes that the model that generated the logging data had the Configuration Parameters > Data Import/Export > Signal logging name set to logsout.

1 Load the MAT-file.
2 Convert logsout to a dataset called myModel_dataset. (The elements information will be different for your data.)

   dataset = logsout.convertToDataset('myModel_Dataset')

   dataset =
   Simulink.SimulationData.Dataset
   Package: Simulink.SimulationData

   Characteristics:    Name: 'myModel_Dataset'
                      Total Elements: 2

   Elements:
      1: 'x1'
      2: 'x2'

   -Use get or getElement to access elements by index or name.
   -Use addElement or setElement to add or modify elements.

Methods, Superclasses
See Also

Topics
“Migrate Scripts That Use Legacy ModelDataLogs API”
“Export Signal Data Using Signal Logging”

Introduced in R2011a
Simulink.restoreDiagnostic

Restore diagnostic warnings to a specific block, subsystem, or model

Syntax

Simulink.restoreDiagnostic(source)
Simulink.restoreDiagnostic(source, message_id)
Simulink.restoreDiagnostic(diagnostic)
Simulink.restoreDiagnostic(system, 'FindAll', 'on')

Description

Simulink.restoreDiagnostic(source) restores all suppressed diagnostics associated with the blocks specified by source.

Simulink.restoreDiagnostic(source, message_id) restores all instances of message_id on the blocks specified by source.

Simulink.restoreDiagnostic(diagnostic) restores the suppressed diagnostics associated with MSLDiagnostic object diagnostic.

Simulink.restoreDiagnostic(system, 'FindAll', 'on') restores all suppressed diagnostics associated with the system specified by system.

Examples

Restore All Diagnostics for a Specified Block

Using the model from “Suppress Diagnostic Messages Programmatically”, restore all suppressed diagnostics on a specified block.

Create a cell array of message identifiers. Use the Simulink.suppressDiagnostic function to suppress these multiple diagnostics from the Constant block, one.
diags = {'SimulinkFixedPoint:util:fxpParameterPrecisionLoss', ...
'SimulinkFixedPoint:util:fxpParameterUnderflow'};
Simulink.suppressDiagnostic('Suppressor_CLI_Demo/one', diags);

Remove all suppressions and restore diagnostics to the block.
Simulink.restoreDiagnostic('Suppressor_CLI_Demo/one');

**Restore a Diagnostic for a Specified Block**

Using the model from “Suppress Diagnostic Messages Programmatically”, restore a suppressed diagnostic on a specified block.

Use the `Simulink.suppressDiagnostic` function to suppress the parameter precision loss warning thrown by the Constant block, `one`.

Simulink.suppressDiagnostic('Suppressor_CLI_Demo/one',...
'SimulinkFixedPoint:util:fxpParameterPrecisionLoss');

Remove the suppression and restore diagnostics to the block.

Simulink.restoreDiagnostic('Suppressor_CLI_Demo/one',...
'SimulinkFixedPoint:util:fxpParameterPrecisionLoss');

**Restore All Diagnostics for a Specified System**

Using the model from “Suppress Diagnostic Messages Programmatically”, restore all suppressed diagnostics on the specified subsystem.

To restore all diagnostics from a system, use 'FindAll', 'on' to search within the system hierarchy. Specify the system or system handle within which to search.

Simulink.restoreDiagnostic('Suppressor_CLI_Demo/Convert',...
'FindAll', 'On');

**Input Arguments**

*source* — Block or model object throwing diagnostic

block path | block handle
The source of the diagnostic, specified as a block path, block handle, cell array of block paths, or cell array of block handles.

To get the block path, use the \texttt{gcb} function.

To get the block handle, use the \texttt{getSimulinkBlockHandle} function.

Data Types: char | cell

\textbf{message\_id — message identifier of diagnostic}

message identifier | cell array of message identifiers

Message identifier of the diagnostic, specified as a character vector or a cell array of character vectors. You can find the message identifier of warnings and errors thrown during simulation by accessing the \texttt{ExecutionInfo} property of the \texttt{Simulink.SimulationMetadata} object associated with a simulation. You can also use the \texttt{lastwarn} function.

Data Types: char | cell

\textbf{system — Name of subsystem}

character vector

The subsystem name, or handle, specified as a character vector.

Data Types: char

\textbf{diagnostic — Diagnostic object}

MSLDiagnostic object

Diagnostic specified as an MSLDiagnostic object. Access the MSLDiagnostic object through the \texttt{ExecutionInfo} property of the \texttt{Simulink.SimulationMetadata} object.

Data Types: struct

\textbf{See Also}

\texttt{Simulink.SuppressDiagnostic} | \texttt{Simulink.SuppressDiagnostic.restore} | \texttt{Simulink.getSuppressedDiagnostics} | \texttt{Simulink.suppressDiagnostic}

\textbf{Topics}

“Suppress Diagnostic Messages Programmatically”
Introduced in R2016b
getBlockSimState

**Class:** Simulink.SimState.ModelSimState  
**Package:** Simulink.SimState

(Not recommended) Access SimState of individual Stateflow Chart, MATLAB Function, or S-function block

---

**Note** Simulink.SimState.ModelSimState.getBlockSimState is not recommended. Use get instead.

---

**Syntax**

```matlab
blockSimState = getBlockSimState(x, 'blockpath')
```

**Description**

`blockSimState = getBlockSimState(x, 'blockpath')` returns the SimState of the block specified as `blockpath`. `blockpath` must be either a Stateflow Chart, MATLAB Function, or S-function block. For other types of blocks, see the `loggedStates` property of the Simulink.SimState.ModelSimState class.

**Input Arguments**

*x*

The `x` argument is a `Simulink.SimState.ModelSimState` object.

**blockpath**

The path to the block for which you are requesting the SimState values.
Output Arguments

blockSimState

The SimState of the block specified.

Examples

chartState = getBlockSimState(x, 'mymodel/chart')

To get the SimState of a block that is in a referenced model, specify the full path of the block relative to the root model.

rootModel = 'sldemo_fuelsys_dd';
opt = struct('SaveFinalState','on','SaveCompleteFinalSimState','on','StopTime','1');
simOut = sim(rootModel,opt);
x = simOut.xFinal;
blockPath = 'sldemo_fuelsys_dd/Fuel Rate Controller|sldemo_fuelsys_dd_controller/control_logic';
chartState = getBlockSimState(x,blockPath)

See Also

Simulink.SimState.ModelState.setBlockSimState
**setBlockSimState**

**Class:** Simulink.SimState.ModelSimState  
**Package:** Simulink.SimState

(Not recommended) Set SimState of individual Stateflow Chart, MATLAB Function, or S-function block

---

**Note** Simulink.SimState.ModelSimState.setBlockSimState is not recommended. Use set instead.

---

**Syntax**

```matlab
setBlockSimState(x,'blockpath', blockSimState)
```

**Description**

`setBlockSimState(x,'blockpath', blockSimState)` sets the SimState of the block specified as `blockpath`. `blockpath` must be either a Stateflow Chart, MATLAB Function, or S-function block. For other types of blocks, see the `loggedStates` property of the Simulink.SimState.ModelSimState class.

**Input Arguments**

- **x**
  
  The argument `x` is a `Simulink.SimState.ModelSimState` object.

- **blockpath**
  
  The path to the block for which you are setting the SimState values
**blockSimState**

The SimState of the block specified.

**Examples**

```matlab
newObj = setBlockSimState(obj, 'mymodel/chart', newChartState);
```

**See Also**

`Simulink.SimState.ModelState.getBlockSimState`
Simulink.saveVars

Save workspace variables and their values in MATLAB code format

Syntax

Simulink.saveVars(filename)
Simulink.saveVars(filename, VarNames)
Simulink.saveVars(filename, '-regexp', RegExps)
Simulink.saveVars(filename, Specifications, UpdateOption)
Simulink.saveVars(filename, Specifications, Configuration)
Simulink.saveVars(filename, Specifications, MatlabVer)
[r1, r2] = Simulink.saveVars(filename, Specifications)

Description

Simulink.saveVars(filename) saves all variables in the current workspace for which MATLAB code can be generated to a MATLAB file named filename.m. If MATLAB code cannot be generated for a variable, the variable is saved into a companion MAT-file named filename.mat, and a warning is generated. If either file already exists, it is overwritten. The filename cannot match the name of any variable in the current workspace, and can optionally include the suffix .m. Using Simulink.saveVars has no effect on the contents of any workspace.

Executing the MATLAB file restores the variables saved in the file to the current workspace. If a companion MAT-file exists, code in the MATLAB file loads the MAT-file, restoring its variables also. When both a MATLAB file and a MAT-file exist, do not load the MATLAB file unless the MAT file is available, or an error will occur. Do not load a MAT-file directly, or incomplete data restoration will result. No warning occurs if loading a file overwrites any existing variables.
You can edit a MATLAB file that Simulink.saveVars creates. You can insert comments between or within the MATLAB code sections for saved variables. However, if you later use Simulink.saveVars to update or append to the file, only comments between MATLAB code sections will be preserved. Internal comments should therefore be used only in files that you do not expect to change any further.

You must not edit the header section in the MATLAB file, which comprises the first five comment lines. Simulink does not check that a manually edited MATLAB file is syntactically correct. MathWorks recommends not editing any MATLAB code in the file. You cannot edit a MAT-file and should never attempt to do so.

Simulink.saveVars(filename, VarNames) saves only the variables specified in VarNames, which is a comma-separated list of variable names. You can use the wildcard character * to save all variables that match a pattern. The * matches one or more characters, including non-alphanumeric characters.

Simulink.saveVars(filename, '-regexp', RegExps) saves only variables whose names match one of the regular expressions in RegExps, which is a comma-separated list of expressions. See “Regular Expressions” (MATLAB) for more information. A call to the function can specify both VarNames and -regexprs RegExps, in that order and comma-separated.

Simulink.saveVars(filename, Specifications, UpdateOption) saves the variables described by Specifications (which represents the variable specifications in any of the above syntaxes) as directed by UpdateOption, which can be any one of the following:

- '-create' — Create a new MATLAB file (and MAT-file if needed) as directed by the Specifications. If either file already exists, it is overwritten. This is the default behavior.
- '-update' — Update the existing MATLAB file (and MAT-file if needed) specified by filename by changing only variables that match the Specifications and already exist in any files. The order of the variables in files is preserved. Comments within MATLAB code sections are not preserved.
- '-append' — Update the existing MATLAB file (and MAT-file if needed) specified by filename by:
  - Updating variables that match the Specifications and already exist in the file or files, preserving the existing order in the file or files. Comments within MATLAB code sections are not preserved.
• Appending variables that match the Specifications and do not exist in the file or files by appending the variables to the file or files. These new sections initially have no comments.

Simulink.saveVars(filename, Specifications, Configuration) saves the variables described by Specifications (which represents the variable specifications in any of the above syntaxes) according to the specified Configuration. The Configuration can contain any or all of the following options, in any order, separated by commas if more than one appears:

• '-maxnumel' MaxNum — Limits the number of elements saved for an array to MaxNum, which must be an integer between 1 and 10000. For a character array, the upper limit is set to twice the value that you specify with MaxNum. If an array is larger than MaxNum, the whole array appears in the MAT-file rather than the MATLAB file, generating a warning. Default: 1000

• '-maxlevels' MaxLevels limits the number of levels of hierarchy saved for a structure or cell array to MaxLevels, which must be an integer between 1 and 200. If a structure or cell array is deeper than MaxLevels, the whole entity appears in the MAT-file rather than the MATLAB file, generating a warning. Default: 20

• '-textwidth' TextWidth sets the text wrap width in the MATLAB file to TextWidth, which must be an integer between 32 and 256. Default: 76

• '-2dslice' — Sets two dimensions for 2-D slices that represent n-D (where n is greater than 2) char, logic, or numeric array data. Simulink.saveVars uses the first two dimensions of the n-D array to specify the size of the 2-D slice, unless you supply two positive integer arguments after the -2dslice option. If you specify two integer arguments:
  • The two integers must be positive.
  • The two integers must be less than or equal to the number of dimensions of the n-D array.
  • The second integer must be greater than the first.

Simulink.saveVars(filename, Specifications, MatlabVer) acts as described by Specifications (which represents the specifications after filename in any of the above syntaxes) saving any MAT-file that it creates in the format required by the MATLAB version specified by MatlabVer. Possible values:

• '-v7.3' — 7.3 or later
• '-v7.0' — 7.0 or later
• `-v6' — Version 6 or later
• `-v4' — Any MATLAB version

\[ r1, r2 \] = Simulink.saveVars(filename, Specifications) acts as described by Specifications (which represents the specifications after filename in any of the above syntaxes) and reports what variables it has saved:

• \( r1 \) — A cell array of character vectors. The character vectors name all variables (if any) that were saved to a MATLAB file.
• \( r2 \) — A cell array of character vectors. The character vectors name all variables (if any) that were saved to a MAT-file.

**Input Arguments**

**filename**

The name of the file or names of the files that the function creates or updates. The filename cannot match the name of any variable in the current workspace. The filename can have the suffix .m, but the function ignores it.

**VarNames**

A variable or sequence of comma-separated variables. The function saves only the specified variables to the output file. You can use the wildcard character * to save all variables that match a pattern. The * matches one or more characters, including non-alphanumeric characters.

**'-regexp', RegExps**

After the keyword, a regular expression or sequence of comma-separated regular expressions. The function saves to the output file only those variables whose names match one of the expressions. See “Regular Expressions” (MATLAB) for more information A call to the function can specify both **VarNames** and **'-regexp' RegExps**, in that order and comma-separated.

**UpdateOption**

Any of three keywords that control the action of the function. The possible values are:

• `'create'` — Create a new MATLAB file (and MAT-file if needed) as directed by the Specifications.
• `-update' — Update the existing MATLAB file (and MAT-file if needed) specified by `filename` by changing only variables that match the `Specifications` and already exist in the file or files. The order of the variables in the file or files is preserved.

• `-append' — Update the existing MATLAB file (and MAT-file if needed) specified by `filename` by:
  
  • Updating variables that match the `Specifications` and already exist in the file or files, preserving the existing order in the file or files.
  
  • appending variables that match the `Specifications` and do not exist in the file or files by appending the variables that match the `Specifications` to the file or files.

**Default:** `-create`

**Configuration**

Any or all of the following options, in any order, separated by commas if more than one appears:

• `-maxnumel` `MaxNum` — Limits the number of elements saved for an array to `MaxNum`, which must be an integer between 0 and 10000. If an array is larger than that, the whole array appears in the MAT-file rather than the MATLAB script file, generating a warning. Default: 1000

• `-maxlevels` `MaxLevels` — Limits the number of levels saved for a structure or cell array to `MaxLevels`, which must be an integer between 0 and 200. If a structure or cell array is deeper than that, the whole entity appears in the MAT-file rather than the MATLAB script file, generating a warning. Default: 20

• `-textwidth` `TextWidth` — Sets the text wrap width in the MATLAB script file to `TextWidth`, which must be an integer between 32 and 256. Default: 76

• `-2dslice' — Sets two dimensions for 2-D slices that represent n-D (where n is greater than 2) arrays of char, logic, or numeric data. Using the `-2dslice' option produces more readable generated code that is consistent with how MATLAB displays n-D array data.

`Simulink.saveVars` uses the first two dimensions of the n-D array to specify the size of the 2-D slice, unless you supply two positive integer arguments after the `-2dslice` option. If you specify two integer arguments:

• The two integers must be positive.
• The two integers must be less than or equal to the number of dimensions of the n-D array.
• The second integer must be greater than the first.

**Note** You can use the MATLAB Preferences to change the defaults for the -maxnumel, -maxlevels, '-2dslice', and -textwidth configuration options. In the **Workspace** pane, use the options in the **Saving variables as MATLAB script files** group.

**MatlabVer**

Specifies the MATLAB version whose syntax will be used by any MAT-file saved by the function.

- ' -v7.3' — 7.3 or later
- ' -v7.0' — 7.0 or later
- ' -v6' — Version 6 or later
- ' -v4' — Any MATLAB version

**Default:** ' -v7.3'

**Output Arguments**

**r1**

A list of the names of all variables (if any) that were saved to a MATLAB file.

**r2**

A list of the names of all variables (if any) that were saved to a MAT-file.

**Examples**

Define some base workspace variables, then save them all to a new MATLAB file named **MyVars.m** using the default values for all input arguments except the **filename**.

```matlab
a = 1;
b = 2.5;
```
c = 'A string';
d = {a, b, c};
Simulink.saveVars('MyVars');

Define additional base workspace variables, then append them to the existing file
MyVars.m without changing the values previously saved in the file:

K = Simulink.Parameter;
MyType = fixdt(1,16,3);
Simulink.saveVars('MyVars', '-append', 'K', 'MyType');

Update the variables V1 and V2 with their values in a MATLAB file, or for any whose value
cannot be converted to MATLAB code, in a MAT-file. The file must already exist. Any array
with more than 10 elements will be saved to a MAT-file that can be loaded on any version
of MATLAB. The return argument r1 lists the names of any variables saved to a MATLAB
file; r2 lists any saved to a MAT-file.

[r1, r2] = Simulink.saveVars('MyFile', 'V1', 'V2', '-update',
'-maxnumel', 10, '-v4');

Specify a 2-D slice for the output of the my3Dtable 3-D array. Specify that the 2-D slice
expands along the first and third dimensions:

my3DTable = zeros(3, 4, 2, 'single');
Simulink.saveVars('mfile.m', 'my3DTable', '-2dslice', 1, 3);

The generated MATLAB code is:

my3DTable = zeros(3, 4, 2, 'single');
my3DTable (:,1,:) = single ( ...
[1 13;
 5 17;
 9 21]);
my3DTable (:,2,:) = single( ... [2 14;
 6 18;
10 22]);
my3DTable (:,3,:) = single( ... [3 15;
 7 19;
11 23]);
my3DTable (:,4,:) = single( ... [4 16;
 8 20;
12 24]);
Limitations

The Simulink.saveVars function:

• Does not preserve shared references.
• Ignores dynamic properties of objects.
• Saves the following to the MAT-file although they could appear in the MATLAB file:
  • Simulink.ConfigSet objects with custom target components.
    (Use the Simulink.ConfigSet method saveAs instead.)
  • Simulink.Timeseries and Simulink.ModelDataLogs objects.

If you save many variables, the generated MATLAB file can contain many lines of code and take a long time to execute. To avoid the long execution time, consider these alternatives:

• Permanently store variables in a data dictionary instead of using Simulink.saveVars. A data dictionary also provides more tools for managing variables. See “Determine Where to Store Variables and Objects for Simulink Models”.
• Save variables in a MAT-file by using the save function.

Tips

• If you do not need to save variables in an easily-understood form, see the save function.
• If you need to save only bus objects, use the Simulink.Bus.save function.
• If you need to save only a configuration set, use the Simulink.ConfigSet.saveAs method.

See Also


Introduced in R2010a
Simulink.sdi.addToList

Package: Simulink.sdi

Add one or more signals to existing run

Syntax

sigIDs = Simulink.sdi.addToList(runID,'vars',var,var2,...,varn)
sigIDs = Simulink.sdi.addToList(runID,'namevalue',sourceNames, dataValues)

Description

sigIDs = Simulink.sdi.addToList(runID,'vars',var,var2,...,varn) adds the data in the variables var, var2, ..., varn to the run corresponding to the runID and returns the signal IDs for the signals added to the run.

sigIDs = Simulink.sdi.addToList(runID,'namevalue',sourceNames, dataValues) adds the data in the cell array dataValues to the run corresponding to the runID and returns the signal IDs for the signals added to the run. The sourceNames argument specifies names to use for the source of the data in dataValues in the signal metadata.

Examples

Add Workspace Data to Run

This example shows how to use Simulink.sdi.addToList to add workspace data to a run in the Simulation Data Inspector.

Generate Workspace Data

Generate workspace data to add to a simulation run in place of measured data, input data, or any other data that you want to associate with the simulation.
time = linspace(0, 60, 201);
cos_vals = 2*cos(2*pi/6.8*time);
cos_ts = timeseries(cos_vals, time);
cos_ts.Name = 'cosine';

**Simulate Model**

Simulate the slexAircraftExample model to create a run containing the simulation outputs.

load_system('slexAircraftExample');
sim('slexAircraftExample','SaveFormat','Dataset');

**Add Workspace Data to Simulation Run**

Add the workspace data to the run. Then, view the results in the Simulation Data Inspector.

% Get run ID
count = Simulink.sdi.getRunCount;
runID = Simulink.sdi.getRunIDByIndex(count);

% Add data to run
sigIDs = Simulink.sdi.addToRun(runID,'vars',cos_ts);

Simulink.sdi.view

**Add States Data to a Run**

The model slexAircraftExample is configured to log outputs, states, and time data. The output data automatically logs to the Simulation Data Inspector as well as the base workspace, but the states data does not. To bring the states data into the Simulation Data Inspector, you can record the data, or you can add it to the run created by simulating the model. This example shows how to add logged states data to a Simulation Data Inspector run programmatically.

**Simulate the Model and Get States Data**

Simulate the model using the sim function with 'ReturnWorkspaceOutputs' set to 'on'. Select the states data, xout, from the simulation outputs.

load_system('slexAircraftExample')
simOut = sim('slexAircraftExample','ReturnWorkspaceOutputs','on',...
'SaveFormat','Dataset');

% Get states data from simulation output
states = simOut.xout;

**Get the Run ID**

Because the outputs data automatically logs to the Simulation Data Inspector, a run is created upon simulating slexAircraftExample. Get the run ID for the run using the Simulation Data Inspector programmatic interface.

runIDs = Simulink.sdi.getAllRunIDs;
runID = runIDs(end);

**Add States Data to the Run**

Add the states data to the run with the output data.

sigIDs = Simulink.sdi.addToRun(runID,'namevalue',{'States'},{states});

**Input Arguments**

**runID** — Run ID to add data to

scalar

Run ID for the signal you want to add data to. The Simulation Data Inspector assigns a unique run ID when it creates a run. You can get the run ID for your run using Simulink.sdi.getAllRunIDs and Simulink.sdi.getRunIDByIndex.

**var** — Data to add to run

variable

Workspace data to add to the run. Simulink.sdi.addToRun supports all loading and logging data formats, including timeseries and Simulink.SimulationData.Dataset. Provide one or more var inputs when you specify 'vars' as the second argument.

Example: myData

**sourceNames** — Cell array of names for signal metadata

cell array of character vectors
Names to use as the source of the data in the metadata for the added signals. Provide a `sourceNames` input when you specify 'namevalue' as the second argument.

Example: {'speed','position'}

**dataValues** — Cell array of data to add to run  
cell array

Cell array of data to add to the run. Provide a `dataValues` input when you specify 'namevalue' as the second argument.

Example: {sig1,sig2}

**Output Arguments**

**sigIDs** — Matrix containing signal IDs for added signals  
matrix

Matrix of signal IDs for signals added to the run.

**See Also**

Simulink.sdi.Run | Simulink.sdi.copyRun | Simulink.sdi.createRun | Simulink.sdi.createRunOrAddToStreamedRun | add

**Topics**

“Inspect and Compare Data Programmatically”

**Introduced in R2011b**
Simulink.sdi.cleanupWorkerResources

Clean up worker repositories

Syntax

Simulink.sdi.cleanupWorkerResources

Description

Simulink.sdi.cleanupWorkerResources removes redundant data from each parallel worker repository file used by the Simulation Data Inspector. Call this function while worker pools are running. The Simulation Data Inspector automatically cleans up repository files when you close the worker pool.

Examples

Access Data from a Parallel Simulation

This example executes parallel simulations of the model slexAircraftExample with different input filter time constants and shows several ways to access the data using the Simulation Data Inspector programmatic interface.

Setup

Start by ensuring the Simulation Data Inspector is empty and Parallel Computing Toolbox support is configured to import runs created on local workers automatically. Then, create a vector of filter parameter values to use in each simulation.

% Make sure the Simulation Data Inspector is empty, and PCT support is enabled.
Simulink.sdi.clear
Simulink.sdi.enablePCTSupport('local')

% Define Ts values
Ts_vals = [0.01, 0.02, 0.05, 0.1, 0.2, 0.5, 1];
**Initialize Parallel Workers**

Use gcp to create a pool of local workers to run parallel simulations if you don't already have one. In an spmd code block, load the slexAircraftExample model and select signals to log. To avoid data concurrency issues using sim in parfor, create a temporary directory for each worker to use during simulations.

```matlab
p = gcp;
Starting parallel pool (parpool) using the 'local' profile ... connected to 4 workers.
spmd

% Load system and select signals to log
load_system('slexAircraftExample')
Simulink.sdi.markSignalForStreaming('slexAircraftExample/Pilot', 1, 'on')
Simulink.sdi.markSignalForStreaming('slexAircraftExample/Aircraft Dynamics Model', 4, 'on')

% Create temporary directory on each worker
workDir = pwd;
addpath(workDir)
tempDir = tempname;
mkdir(tempDir)
cd(tempDir)
end

**Run Parallel Simulations**

Use parfor to run the seven simulations in parallel. Select the value for Ts for each simulation, and modify the value of Ts in the model workspace. Then, run the simulation and build an array of Simulink.sdi.WorkerRun objects to access the data with the Simulation Data Inspector. After the parfor loop, use another spmd segment to remove the temporary directories from the workers.

```matlab
parfor index = 1:7

% Select value for Ts
Ts_val = Ts_vals(index);

% Change the filter time constant and simulate
modelWorkspace = get_param('slexAircraftExample','modelworkspace');
modelWorkspace.assignin('Ts',Ts_val)
```
sim('slexAircraftExample')

% Create a worker run for each simulation
workerRun(index) = Simulink.sdi.WorkerRun.getLatest
end

spmd

% Remove temporary directories
cd(workDir)
rmdir(tempDir, 's')
rmpath(workDir)
end

Get Dataset Objects from Parallel Simulation Output

The `getDataset` method puts the data from a `WorkerRun` into a `Dataset` object so you can easily post-process.

ds(7) = Simulink.SimulationData.Dataset;

for a = 1:7
    ds(a) = workerRun(a).getDataset;
end

ds(1)

ans = Simulink.SimulationData.Dataset '' with 2 elements

<table>
<thead>
<tr>
<th>Name</th>
<th>BlockPath</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 [1x1 Signal]</td>
<td>alpha, rad ...rcraftExample/Aircraft Dynamics Model</td>
</tr>
<tr>
<td>2 [1x1 Signal]</td>
<td>Stick slexAircraftExample/Pilot</td>
</tr>
</tbody>
</table>

- Use braces {} to access, modify, or add elements using index.

Get DatasetRef Objects from Parallel Simulation Output

For big data workflows, use the `getDatasetRef` method to reference the data associated with the `WorkerRun`.

for b = 1:7
    datasetRef(b) = workerRun(b).getDatasetRef;
end
end

datasetRef(1)

ans =
    DatasetRef with properties:
        Name: 'Run 3: slexAircraftExample'
        Run: [1x1 Simulink.sdi.Run]
    numElements: 2

Process Parallel Simulation Data in the Simulation Data Inspector

You can also create local Run objects to analyze and visualize your data using the Simulation Data Inspector API. This example adds a tag indicating the filter time constant value for each run.

```matlab
for c = 1:7
    Runs(c) = workerRun(c).getLocalRun;
    Ts_val_str = num2str(Ts_vals(c));
    desc = strcat('Ts = ', Ts_val_str);
    Runs(c).Description = desc;
    Runs(c).Name = strcat('slexAircraftExample run Ts=', Ts_val_str);
end
```

Clean Up Worker Repositories

Clean up the files used by the workers to free up disk space for other simulations you want to run on your worker pool.

`Simulink.sdi.cleanupWorkerResources`

See Also

`Simulink.sdi.WorkerRun` | `Simulink.sdi.isPCTSupportEnabled`

Topics

“Inspect and Compare Data Programmatically”
Introduced in R2017b
Simulink.sdi.clear

**Package:** Simulink.sdi

Clear all data from the Simulation Data Inspector

**Syntax**

Simulink.sdi.clear

**Description**

Simulink.sdi.clear clears all plotted signals and deletes all data from the Simulation Data Inspector. The Simulink.sdi.clear function does not affect preferences or settings you have configured in the Simulation Data Inspector. Use the Simulink.sdi.clearPreferences function to reset the Simulation Data Inspector preferences to their default values. Use the Simulink.sdi.clearAllSubPlots function to clear all plotted signals without deleting any data from the Simulation Data Inspector.

**Examples**

**Save a Simulation Data Inspector Session**

This example shows how to create, save, and load a Simulation Data Inspector session. The example uses data logging to populate the Simulation Data Inspector with data and then uses the Simulation Data Inspector's programmatic interface to create plots to visualize the data. After saving the data and visualization settings in a session, the Simulation Data Inspector repository is emptied in order to demonstrate how to load the session.
Create Simulation Data

This example logs the Stick, alpha, rad, and q, rad/sec signals to generate simulation data using the model slexAircraftExample and creates two runs. The first uses a sine input, and the second has a square wave input.

% Ensure you start with an empty Simulation Data Inspector repository
Simulink.sdi.clear

% Load system
load_system('slexAircraftExample')

% Configure signals to log
Simulink.sdi.markSignalForStreaming('slexAircraftExample/Pilot',1,'on')
Simulink.sdi.markSignalForStreaming('slexAircraftExample/Aircraft Dynamics Model',3,'on')
Simulink.sdi.markSignalForStreaming('slexAircraftExample/Aircraft Dynamics Model',4,'on')

% Change Pilot signal to sine
set_param('slexAircraftExample/Pilot','WaveForm','sine')

% Simulate model
sim('slexAircraftExample')

% Change Pilot signal to square
set_param('slexAircraftExample/Pilot','WaveForm','square')

% Simulate model
sim('slexAircraftExample')

Access Simulation Data

Use the Simulation Data Inspector programmatic interface to access the simulation data so you can create plots to visualize the signals.

% Get run objects
runIDs = Simulink.sdi.getAllRunIDs;
sineRunID = runIDs(end-1);
squareRunID = runIDs(end);
sineRun = Simulink.sdi.getRun(sineRunID);
squareRun = Simulink.sdi.getRun(squareRunID);

% Get signal objects
sineOut = sineRun.getSignalByIndex(1);
sineIn = sineRun.getSignalByIndex(3);
squareOut = squareRun.getSignalByIndex(1);
squareIn = squareRun.getSignalByIndex(3);

**Create Plots in the Simulation Data Inspector**

Use the programmatic interface to visualize the signal data from the two simulation runs. You can set the plot layout and plot signals on specific subplots.

```
% Set subplot layout
Simulink.sdi.setSubPlotLayout(2,1)

% Plot sine data on top plot
sineIn.plotOnSubPlot(1,1,true)
sineOut.plotOnSubPlot(1,1,true)

% Plot square wave data on bottom plot
squareIn.plotOnSubPlot(2,1,true)
squareOut.plotOnSubPlot(2,1,true)
```

**Save a Simulation Data Inspector Session**

First, view the plots you just created. Then, save the Simulation Data Inspector session as an MLDATX-file to recover your data along with your preference selections and plots.

```
% View the visualized data in the Simulation Data Inspector
Simulink.sdi.view

% Save the Simulation Data Inspector session
Simulink.sdi.save('myData.mldatx')
```

**Load a Simulation Data Inspector Session**

First, clear the Simulation Data Inspector repository with Simulink.sdi.clear and reset visualization settings with Simulink.sdi.clearPreferences. Then, you can load the session to see how the data and settings are preserved.

```
% Clear Simulation Data Inspector repository and preferences
Simulink.sdi.clear
Simulink.sdi.clearPreferences
```
% Load session file to view data
Simulink.sdi.load('myData.mldatx');

See Also
Simulink.sdi.clearAllSubPlots | Simulink.sdi.clearPreferences |
Simulink.sdi.deleteRun | Simulink.sdi.deleteSignal

Topics
“Inspect and Compare Data Programmatically”

Introduced in R2011b
Simulink.sdi.clearAllSubPlots

Clear plotted signals from all subplots in the Simulation Data Inspector

Syntax

Simulink.sdi.clearAllSubPlots

Description

Simulink.sdi.clearAllSubPlots clears plotted signals from all subplots in the Simulation Data Inspector, including any subplots with plotted data that are not visible in the current layout. Clearing plotted signals using the Simulink.sdi.clearAllSubPlots function does not delete data from the Simulation Data Inspector. To clear plotted signals and delete data from the Simulation Data Inspector, use the Simulink.sdi.clear function.

Examples

Clear Plotted Signals from All Subplots

You can programmatically clear plotted signals from all subplots in the Simulation Data Inspector. For example, clear plotted data at the start of a script that creates plots.

Simulink.sdi.clearAllSubPlots

Plot Signals from a Simulation Run

This example demonstrates how to access the Simulink.sdi.Run object for a Simulation Data Inspector run created by logging signals. From the Simulink.sdi.Run object you can get Simulink.sdi.Signal objects that you can use to plot the data.
Create a Simulation Run and Access the Run Object

The sldemo_fuelsys model includes several signals that are marked for logging. To create a simulation run containing the logged data, simulate the model.

\[ \text{sim('ex_vdp');} \]

The Simulation Data Inspector keeps track of runs by assigning each run created by simulation, importing data, or opening a session a unique numeric ID. To access the run object for the simulation just performed, use the \texttt{Simulink.sdi.getAllRunIDs} function and take the last run ID in the returned vector.

\[ \text{runIDs} = \text{Simulink.sdi.getAllRunIDs;} \]
\[ \text{runID} = \text{runIDs(end);} \]

Once you have the run ID for the run of interest, you can use the \texttt{Simulink.sdi.getRun} function to get the \texttt{Simulink.sdi.Run} object corresponding to the run. You can use the \texttt{Run} object to check metadata associated with the run, including the number of signals in the run.

\[ \text{vdpRun} = \text{Simulink.sdi.getRun(runID);} \]
\[ \text{vdpRun.SignalCount} \]
\[ \text{ans} = \text{int32} \]
\[ 2 \]

Plot Data Using Signal Objects

Use the \texttt{getSignalByIndex} function to access signals of interest from the Run object, \texttt{fuelRun}.

\[ \text{signal1} = \text{getSignalByIndex(vdpRun,1);} \]
\[ \text{signal2} = \text{getSignalByIndex(vdpRun,2);} \]

Use the \texttt{Simulink.sdi.setSubPlotLayout} function to specify a 3-by-1 layout.

\[ \text{Simulink.sdi.setSubPlotLayout(2,1);} \]

Before plotting the data, use the \texttt{Simulink.sdi.clearAllSubPlots} function to clear any data that is already plotted.

\[ \text{Simulink.sdi.clearAllSubPlots} \]
Plot one signal on each subplot. To plot signals on the first subplot, you can set the checked property for the signal. To plot signals on subplots other than the first subplot, use the `plotOnSubPlot` function.

```matlab
signal1.Checked = true;
plotOnSubPlot(signal2,2,1,true);
```

**View the Plotted Data**

To view the plots you just created, open the Simulation Data Inspector using the `Simulink.sdi.view` function.
See Also

Simulink.sdi.Signal | Simulink.sdi.clear |
Simulink.sdi.clearPreferences | Simulink.sdi.setSubPlotLayout |
plotOnSubPlot

Topics

“Create Plots Using the Simulation Data Inspector”

Introduced in R2019b
Simulink.sdi.clearPreferences

Return Simulation Data Inspector preferences to default settings

Syntax

Simulink.sdi.clearPreferences

Description

Simulink.sdi.clearPreferences reverts all Simulation Data Inspector preferences to their default values.

You can configure these preferences in the Simulation Data Inspector programmatically:

- Preferences:
  - Run naming rule — Simulink.sdi.setRunNamingRule
  - Subplot layout — Simulink.sdi.setSubPlotLayout
  - Signal grouping within a run — Simulink.sdi.setTableGrouping
- Time plot settings:
  - Border display — Simulink.sdi.setBorderOn
  - Grid display — Simulink.sdi.setGridOn
  - Data markers display — Simulink.sdi.setMarkersOn
  - Tick mark labels display — Simulink.sdi.setTickLabelsDisplay
  - Tick mark position — Simulink.sdi.setTicksPosition
- Archive preferences:
  - Archive behavior — Simulink.sdi.setAutoArchiveMode
  - Archive size limit — Simulink.sdi.setArchiveRunLimit
Examples

Restore All Simulation Data Inspector Preferences to Default Values

You can restore default values to all Simulation Data Inspector preferences programmatically using the `Simulink.sdi.clearPreferences` function.

`Simulink.sdi.clearPreferences`

Modify Run Naming Rule Then Restore Default

This example shows how to use the Simulation Data Inspector API to modify the Simulation Data Inspector run naming rule, check a run’s name, restore default preferences, and check the run naming rule.

```matlab
% Load model
load_system('sldemo_fuelsys')

% Modify run naming rule
Simulink.sdi.setRunNamingRule('<model_name> Run <run_index>')</n
% Simulate system
sim('sldemo_fuelsys')

% Check run name
runIDs = Simulink.sdi.getAllRunIDs;
runID = runIDs(end);
fuelRun = Simulink.sdi.getRun(runID);
fuelRun.name

ans =
'sldemo_fuelsys Run 1'

% Clear preferences to reset the run naming rule
Simulink.sdi.clearPreferences

% Check run naming rule
Simulink.sdi.getRunNamingRule

ans =
'Run <run_index>: <model_name>'
```
See Also
Simulink.sdi.clear | Simulink.sdi.clearAllSubPlots

Topics
“Inspect and Compare Data Programmatically”
“Organize Your Simulation Data Inspector Workspace”

Introduced in R2017a
Simulink.sdi.close

Package: Simulink.sdi

Close the Simulation Data Inspector

Syntax

Simulink.sdi.close
Simulink.sdi.close('filename')

Description

Simulink.sdi.close closes the Simulation Data Inspector.

Simulink.sdi.close('filename') closes the Simulation Data Inspector and saves the data in the file, filename.

Examples

Close the Simulation Data Inspector from the Command Line

You can close the Simulation Data Inspector from the MATLAB command line when you have finished inspecting and analyzing your data.

Simulink.sdi.close

Create a Run and View the Data

This example shows how to create a run, add data to it, and then view the data in the Simulation Data Inspector.
Create Data for the Run

This example creates timeseries objects for a sine and a cosine. To visualize your data, the Simulation Data Inspector requires at least a time vector that corresponds with your data.

```matlab
% Generate timeseries data
time = linspace(0, 20, 100);
sine_vals = sin(2*pi/5*time);
sine_ts = timeseries(sine_vals, time);
sine_ts.Name = 'Sine, T=5';

cos_vals = cos(2*pi/8*time);
cos_ts = timeseries(cos_vals, time);
cos_ts.Name = 'Cosine, T=8';
```

Create a Simulation Data Inspector Run and Add Your Data

To give the Simulation Data Inspector access to your data, use the `create` method and create a run. This example modifies some of the run's properties to help identify the data. You can easily view run and signal properties with the Simulation Data Inspector.

```matlab
% Create a run
sinusoidsRun = Simulink.sdi.Run.create;
sinusoidsRun.Name = 'Sinusoids';
sinusoidsRun.Description = 'Sine and cosine signals with different frequencies';

% Add timeseries data to run
sinusoidsRun.add('vars', sine_ts, cos_ts);
```

Plot Your Data Using the `Simulink.sdi.Signal` Object

The `getSignalByIndex` method returns a `Simulink.sdi.Signal` object that can be used to plot the signal in the Simulation Data Inspector. You can also programmatically control aspects of the plot's appearance, such as the color and style of the line representing the signal. This example customizes the subplot layout and signal characteristics.

```matlab
% Get signal, modify its properties, and change Checked property to true
sine_sig = sinusoidsRun.getSignalByIndex(1);
sine_sig.LineColor = [0 0 1];
sine_sig.LineDashed = '-.';
sine_sig.Checked = true;
```
% Add another subplot for the cosine signal
Simulink.sdi.setSubPlotLayout(2, 1);

% Plot the cosine signal and customize its appearance
cos_sig = sinusoidsRun.getSignalByIndex(2);
cos_sig.LineColor = [0 1 0];
cos_sig.plotOnSubPlot(2, 1, true);

% View the signal in the Simulation Data Inspector
Simulink.sdi.view

Close the Simulation Data Inspector and Save Your Data
Simulink.sdi.close('sinusoids.mat')

See Also
Simulink.sdi.clear | Simulink.sdi.clearPreferences | Simulink.sdi.save | Simulink.sdi.view

Topics
“Inspect and Compare Data Programmatically”
“Save and Share Simulation Data Inspector Data and Views”

Introduced in R2013b
Simulink.sdi.compareRuns

Package: Simulink.sdi

Compare the data in two simulation runs

Syntax

diffResult = Simulink.sdi.compareRuns(runID1,runID2)
diffResult = Simulink.sdi.compareRuns(runID1,runID2,Name,Value)

Description

diffResult = Simulink.sdi.compareRuns(runID1,runID2) compares the data in the runs corresponding to runID1 and runID2, returning the result in the Simulink.sdi.DiffRunResult object, diffResult.

diffResult = Simulink.sdi.compareRuns(runID1,runID2,Name,Value) compares the simulation runs corresponding to runID1 and runID2 using the options specified by one or more Name,Value pair arguments and returns the comparison result in the Simulink.sdi.DiffRunResult object, diffResult. For more information about how the Simulation Data Inspector aligns signals for comparison, see “How the Simulation Data Inspector Compares Data”.

Examples

Compare Simulation Data Inspector Runs Programmatically

This example shows how to compare runs of simulation data and analyze the comparison results with the Simulation Data Inspector programmatic interface.

Generate Runs of Simulation Data

Simulate the model with different Desired relative slip values to create runs of simulation data to analyze with the Simulation Data Inspector programmatic interface.
% Open model
load_system('ex_sldemo_absbrake')

% Set the desired slip ratio to 0.24 and simulate
set_param('ex_sldemo_absbrake/Desired relative slip','Value','0.24')
out_1 = sim('ex_sldemo_absbrake');

% Change the desired slip ratio to 0.25 and simulate
set_param('ex_sldemo_absbrake/Desired relative slip','Value','0.25')
out_2 = sim('ex_sldemo_absbrake');

**Compare Runs with a Global Tolerance**

Get the run IDs for the runs you just created with the `Simulink.sdi.getAllRunIDs` function. Then, compare the runs using a global relative tolerance and a global time tolerance to analyze whether your data meets specifications.

% Get run IDs for last two runs
runIDs = Simulink.sdi.getAllRunIDs;
runID1 = runIDs(end - 1);
runID2 = runIDs(end);

% Compare runs
runResult = Simulink.sdi.compareRuns(runID1,runID2,'reltol',0.2,'timetol',0.5);

**Create a Plot of a Comparison Result**

Use the `Simulink.sdi.DiffRunResult` object you created in the previous step with `Simulink.sdi.compareRuns` to access the data for the Ww signal result to plot it in a figure.

% Plot the |Ww| signal difference
signalResult_Ww = getResultByIndex(runResult,1);
figure(1)
plot(signalResult_Ww.Diff)
Analyze Simulation Data with Signal Tolerances

You can change tolerance values on a signal-by-signal basis to evaluate the effect of a model parameter change. This example uses the slexAircraftExample model and the Simulation Data Inspector to evaluate the effect of changing the time constant for the low-pass filter following the control input.

Setup

Load the model, and mark the q, rad/sec and alpha, rad signals for logging. Then, simulate the model to create the baseline run.
% Load example model
load_system('slexAircraftExample')

% Mark the q, rad/sec and alpha, rad signals for logging
Simulink.sdi.markSignalForStreaming('slexAircraftExample/Aircraft Dynamics Model',3,'on')
Simulink.sdi.markSignalForStreaming('slexAircraftExample/Aircraft Dynamics Model',4,'on')

% Simulate system
out_1 = sim('slexAircraftExample');

**Modify Model Parameter**
Modify the model parameter Ts in the model workspace to change the time constant of
the input low-pass filter.

% Change input filter time constant
modelWorkspace = get_param('slexAircraftExample','modelworkspace');
assignin(modelWorkspace,'Ts',1)

% Simulate again
out_2 = sim('slexAircraftExample');

**Compare Runs and Inspect Results**
Use the Simulink.sdi.compareRuns function to compare the data from the simulations. Then, inspect the match property of the signal result to see whether the signals fell within the default tolerance of 0.

% Get run data
runIDs = Simulink.sdi.getAllRunIDs;
runID1 = runIDs(end - 1);
runID2 = runIDs(end);

% Compare runs
diffRun1 = Simulink.sdi.compareRuns(runID1,runID2);

% Get signal result
sig1Result1 = getResultByIndex(diffRun1,1);
sig2Result1 = getResultByIndex(diffRun1,2);

% Check whether signals matched
sig1Result1.Status
ans =
OutOfTolerance
sig2Result1.Status
ans =
OutOfRange

**Compare Runs with Signal Tolerances**

The signals did not match within the default tolerance of 0. To further analyze the effect of the time constant change, add signal tolerances to the comparison with the baseline signal properties to determine the tolerance required for a pass. This example uses a combination of time and absolute tolerances.

% Get signal object for sigID1
run1 = Simulink.sdi.getRun(runID1);
sigID1 = getSignalIDByIndex(run1,1);
sigID2 = getSignalIDByIndex(run1,2);

sig1 = Simulink.sdi.getSignal(sigID1);
sig2 = Simulink.sdi.getSignal(sigID2);

% Set tolerances for q, rad/sec
sig1.AbsTol = 0.1;
sig1.TimeTol = 0.6;

% Set tolerances for alpha, rad
sig2.AbsTol = 0.2;
sig2.TimeTol = 0.8;

% Run the comparison again
diffRun2 = Simulink.sdi.compareRuns(runID1,runID2);
sig1Result2 = getResultByIndex(diffRun2,1);
sig2Result2 = getResultByIndex(diffRun2,2);

% Check the result
sig1Result2.Status
ans =
WithinTolerance

sig2Result2.Status
ans =
WithinTolerance
**Compare Runs with Alignment Criteria**

This example shows how to compare runs using your desired criteria for aligning signals between runs.

**Generate Simulation Data to Compare**

Start by loading the `slexAircraftExample` system and marking signals of interest for logging.

```matlab
% Load the model slexAircraftExample
load_system('slexAircraftExample')

% Mark signals for logging
Simulink.sdi.markSignalForStreaming('slexAircraftExample/Pilot',1,'on')
Simulink.sdi.markSignalForStreaming('slexAircraftExample/Aircraft Dynamics Model',4,'on')
```

This example runs two simulations of the `slexAircraftExample` model with different values of the variable `Ts`. Use `get_param` to access the model workspace for the `slexAircraftExample` model. Then, assign a value for the variable and run the first simulation.

```matlab
% Get the model workspace and set the value of variable Ts
modelWorkspace = get_param('slexAircraftExample','modelworkspace');
modelWorkspace.assignin('Ts',0.1);

% Configure model "slexAircraftExample" for logging and simulate
out_1 = sim('slexAircraftExample');
```

Change the value of `Ts` and run the second simulation. Then, use the Simulation Data Inspector programmatic interface to access the run IDs for the two simulation runs.

```matlab
% Modify the value of model workspace variable Ts
modelWorkspace.assignin('Ts',0.2);

% Simulate again with the new value of Ts
out_2 = sim('slexAircraftExample');

% Get run IDs for simulations
runIDs = Simulink.sdi.getAllRunIDs;
runID1 = runIDs(end-1);
runID2 = runIDs(end);
```
Define Alignment Criteria for the Comparison

Before running the comparison, define how you want the Simulation Data Inspector to align the signals between the runs. This example aligns signals by their name, then by their block path, and then by their Simulink identifier.

```matlab
% Define the alignment criteria for the comparison to align signals by
% name, then by block path, then by SID
alignMethods = [Simulink.sdi.AlignType.SignalName
                Simulink.sdi.AlignType.BlockPath
                Simulink.sdi.AlignType.SID];
```

Compare the Runs with the Specified Alignment Criteria

Compare the simulation data in your two runs, using the alignment criteria you specified. The comparison uses a small time tolerance to account for the effect of differences in the step size used by the solver on the transition of the square wave input.

```matlab
% Compare the runs
diffResults = Simulink.sdi.compareRuns(runID1,runID2,'align',alignMethods,...
    'timetol',0.005);
```

Check the Comparison Results for the Aligned Signals

You can use the `getResultByIndex` method to access the comparison results for the aligned signals in the runs you compared.

```matlab
% Check the number of comparisons in the result
numComparisons = diffResults.count;

% Display result for each signal in command window
for k = 1:numComparisons
    resultAtIdx = diffResults.getResultByIndex(k);

    % Get signal IDs for each comparison result
    sigID1 = resultAtIdx.signalID1;
    sigID2 = resultAtIdx.signalID2;

    sig1 = Simulink.sdi.getSignal(sigID1);
    sig2 = Simulink.sdi.getSignal(sigID2);

    % Display whether signals match
    displayStr = 'Signals %s and %s: %s
' ;
```
fprintf(displayStr,sig1.Name,sig2.Name,resultAtIdx.Status);

end

Signals alpha, rad and alpha, rad: OutOfTolerance
Signals Stick and Stick: WithinTolerance

**Input Arguments**

**runID1** — Baseline run identifier  
integer

Numerical identification for the **Baseline** run in the comparison.

**runID2** — Compare to run identifier  
integer

Numerical identification for the **Compare to** run in the comparison.

**Name-Value Pair Arguments**

Specify optional comma-separated pairs of **Name**, **Value** arguments. **Name** is the argument name and **Value** is the corresponding value. **Name** must appear inside quotes. You can specify several name and value pair arguments in any order as  **Name1,** **Value1,**..., **NameN,** **ValueN.**

Example: `'abstol',x,'align',alignOpts`

**align** — Signal alignment options
array

Array specifying alignment algorithms to use for pairing signals from the runs for comparison. The Simulation Data Inspector aligns signals first by the first element in the array, then by the second element in the array, and so on.

<table>
<thead>
<tr>
<th>Value</th>
<th>Aligns By</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Simulink.sdi.AlignType.BlockPath</code></td>
<td>Path to the signal's source block</td>
</tr>
<tr>
<td><code>Simulink.sdi.AlignType.SID</code></td>
<td>Simulink identifier “Locate Diagram Components Using Simulink Identifiers”</td>
</tr>
<tr>
<td><code>Simulink.sdi.AlignType.SignalName</code></td>
<td>Signal name</td>
</tr>
<tr>
<td>Value</td>
<td>Aligns By</td>
</tr>
<tr>
<td>--------------------------</td>
<td>------------------------------------------------</td>
</tr>
<tr>
<td>Simulink.sdi.AlignType.DataSource</td>
<td>Path of the variable in the MATLAB workspace</td>
</tr>
</tbody>
</table>

Example:

```markdown
[Simulink.sdi.AlignType.SignalName, Simulink.sdi.AlignType.SID]
```

specifies signal alignment by name and then by SID.

**abstol — Absolute tolerance for comparison**

0 (default) | scalar

Positive-valued global absolute tolerance used for all signals in the run comparison. For more information about tolerances in the Simulation Data Inspector, see “How the Simulation Data Inspector Compares Data”.

Example: 0.5

Data Types: double

**reltol — Relative tolerance for comparison**

0 (default) | scalar

Positive-valued global relative tolerance used for all signals in the run comparison. The relative tolerance is expressed as a fractional multiplier. For example, 0.1 specifies a 10 percent tolerance. For more information about how the relative tolerance is applied in the Simulation Data Inspector, see “How the Simulation Data Inspector Compares Data”.

Example: 0.1

Data Types: double

**timetol — Time tolerance for comparison**

0 (default) | scalar

Positive-valued global time tolerance used for all signals in the run comparison. Specify the time tolerance with units of seconds. For more information about tolerances in the Simulation Data Inspector, see “How the Simulation Data Inspector Compares Data”.

Example: 0.2

Data Types: double
Output Arguments

diffResult — Comparison results
'Simulink.sdi.diffRunResult'

Simulink.sdi.DiffRunResult object that provides access to comparison results.

See Also
Simulink.sdi.DiffRunResult | Simulink.sdi.DiffRunResult.getResultByIndex | Simulink.sdi.DiffSignalResult | Simulink.sdi.compareSignals | Simulink.sdi.getRunCount | Simulink.sdi.getRunIDByIndex

Topics
“Inspect and Compare Data Programmatically”
“Compare Simulation Data”
“How the Simulation Data Inspector Compares Data”

Introduced in R2011b
Simulink.sdi.compareSignals

Package: Simulink.sdi

Compare data from two signals

Syntax

diff = Simulink.sdi.compareSignals(sigID1,sigID2)

Description

diff = Simulink.sdi.compareSignals(sigID1,sigID2) compares the signals corresponding to the signal IDs sigID1 and sigID2 and returns the results in a Simulink.sdi.DiffSignalResult object. For more information on how the comparison results are computed, see “How the Simulation Data Inspector Compares Data”.

Examples

Compare Signals Within a Simulation Run

This example uses the slexAircraftExample model to demonstrate the comparison of the input and output signals for a control system. The example marks the signals for streaming then gets the run object for a simulation run. Signal IDs from the run object specify the signals to be compared.

% Load model slexAircraftExample and mark signals for streaming
load_system('slexAircraftExample')
Simulink.sdi.markSignalForStreaming('slexAircraftExample/Pilot',1,'on')
Simulink.sdi.markSignalForStreaming('slexAircraftExample/Aircraft Dynamics Model',4,'on')

% Simulate model slexAircraftExample
out = sim('slexAircraftExample');
% Get run IDs for most recent run
allIDs = Simulink.sdi.getAllRunIDs;
runID = allIDs(end);

% Get Run object
aircraftRun = Simulink.sdi.getRun(runID);

% Get signal IDs
signalID1 = getSignalIDByIndex(aircraftRun,1);
signalID2 = getSignalIDByIndex(aircraftRun,2);

if (isValidSignalID(aircraftRun,signalID1))
    % Change signal tolerance
    signal1 = Simulink.sdi.getSignal(signalID1);
    signal1.AbsTol = 0.1;
end

if (isValidSignalID(aircraftRun,signalID1) && isValidSignalID(aircraftRun,signalID2))
    % Compare signals
    sigDiff = Simulink.sdi.compareSignals(signalID1,signalID2);

    % Check whether signal result status
    match = sigDiff.Status
end

match = OutOfTolerance

**Compare Signals from Different Runs**

This example shows how to compare signals from different simulation runs using the Simulation Data Inspector's Simulink.sdi.compareSignals function. When you only have one signal of interest to compare, using a signal comparison returns the Simulink.sdi.diffSignalResult object with the comparison data directly.

**Generate Simulation Data**

Use the slexAircraftExample model to generate simulation runs. Between the runs, change the time constant of the input filter.

% Load example model
load_system('slexAircraftExample')

% Mark the alpha, rad signal for streaming
Simulink.sdi.markSignalForStreaming('slexAircraftExample/Aircraft Dynamics Model',4,'on')

% Simulate system
out_1 = sim('slexAircraftExample');

% Change input filter time constant
modelWorkspace = get_param('slexAircraftExample','modelworkspace');
assignin(modelWorkspace,'Ts',0.2)

% Simulate again
out_2 = sim('slexAircraftExample');

Get Signal IDs for the Signal Comparison

Create run objects using the run IDs, and then use getSignalIDByIndex to get the signal IDs to pass to Simulink.sdi.compareSignals.

% Get run data
runIDs = Simulink.sdi.getAllRunIDs;

runID1 = runIDs(end-1);
runID2 = runIDs(end);

run1 = Simulink.sdi.getRun(runID1);
run2 = Simulink.sdi.getRun(runID2);

sigID1 = getSignalIDByIndex(run1,1);
sigID2 = getSignalIDByIndex(run2,1);

Compare Signals

Compare the signals, and open the Simulation Data Inspector to view the results.

diffResult = Simulink.sdi.compareSignals(sigID1,sigID2);
Simulink.sdi.view

Input Arguments

sigID1 — Signal ID of Baseline signal
scalar
Signal ID for the **Baseline** signal. The Simulation Data Inspector assigns signal and run IDs when you create a run for your data. You can access the signal ID from a Simulink.sdi.Run object with the `getSignalIDByIndex` method.

**sigID2 — Signal ID of Compare to signal**

scalar

Signal ID for the **Compare to** signal. The Simulation Data Inspector assigns signal and run IDs when you create a run for your data. You can access the signal ID from a Simulink.sdi.Run object with the `getSignalIDByIndex` method.

**Output Arguments**

**diff — Comparison results**

'Simulink.sdi.diffSignalResult'

Simulink.sdi.DiffSignalResult object containing the results of the signal comparison.

**See Also**

Simulink.sdi.DiffSignalResult | Simulink.sdi.Run | Simulink.sdi.compareRuns

**Topics**

“Inspect and Compare Data Programmatically”

“Compare Simulation Data”

“How the Simulation Data Inspector Compares Data”

**Introduced in R2011b**
Simulink.sdi.copyRun

Copy a Simulation Data Inspector run

**Syntax**

```matlab
newRunID = Simulink.sdi.copyRun(runID)
[newRunID,runIndex] = Simulink.sdi.copyRun(runID)
[newRunID,runIndex,signalIDs] = Simulink.sdi.copyRun(runID)
```

**Description**

`newRunID = Simulink.sdi.copyRun(runID)` copies the run corresponding to `runID` and returns the run ID for the new run. The new run includes all the simulation data and metadata from the original run. You can modify the copy of the run by adding or deleting signals and metadata while still retaining the original run.

`[newRunID,runIndex] = Simulink.sdi.copyRun(runID)` copies the run corresponding to `runID` and returns the run ID and index in the Simulation Data Inspector repository for the new run. The new run includes all the simulation data and metadata from the original run.

`[newRunID,runIndex,signalIDs] = Simulink.sdi.copyRun(runID)` copies the run corresponding to `runID` and returns the signal IDs for the signals in the new run along with its run ID and index in the Simulation Data Inspector repository. The new run includes all the simulation data and metadata from the original run.

**Examples**

**Compare a Subset of Signals**

This example shows how to use `Simulink.sdi.copyRun` and `Simulink.sdi.deleteSignal` to create a copy of a run that contains a subset of the signals from the original run. You can use the copy to analyze and run comparisons on a
subset of signals while still holding onto the original run that has all of the signals. For example, the model sldemo_fuelsys is configured to log ten signals. To compare the system's responses to different types of failures, you don't need to run the comparison on all of the logged signals. Deleting signals that do not represent the system's response before running the comparison saves processing time and simplifies the view of the results.

**Create Runs**

Load the model sldemo_fuelsys and run simulations to create runs in the Simulation Data Inspector. The first run simulates a failure of the throttle angle sensor, and the second run simulates a failure of the exhaust gas oxygen sensor:

```matlab
load_system('sldemo_fuelsys')
modelWorkspace = get_param('sldemo_fuelsys','modelworkspace');
modelWorkspace.assignin('throttle_sw',0)
modelWorkspace.assignin('ego_sw',1)
sim('sldemo_fuelsys')
modelWorkspace.assignin('throttle_sw',1)
modelWorkspace.assignin('ego_sw',0)
sim('sldemo_fuelsys')
```

**Copy the Run**

Use the Simulation Data Inspector's programmatic interface to get Simulink.sdi.Run objects for the simulations, and then create copies of the runs.

```matlab
% Get runs
runIDs = Simulink.sdi.getAllRunIDs;
runID1 = runIDs(end-1);
runID2 = runIDs(end);
run1 = Simulink.sdi.getRun(runID1);
run2 = Simulink.sdi.getRun(runID2);

% Create a copy of each run, truncRun
[truncRun1,runIndex1,signalIDs1] = Simulink.sdi.copyRun(runID1);
[truncRun2,runIndex2,signalIDs2] = Simulink.sdi.copyRun(runID2);
```

**Delete Signals in Run Copy**

The sldemo_fuelsys model is configured to log the values of the fault switches along with several signals representing the system's response. When you compare the system's
response when a throttle angle sensor fails to its response when an exhaust gas oxygen sensor fails, comparing the fault switch states does not provide new information. Therefore, delete the switch signals before running the comparison to eliminate unnecessary computations.

Simulink.sdi.deleteSignal(signalIDs1(1))
Simulink.sdi.deleteSignal(signalIDs1(3))
Simulink.sdi.deleteSignal(signalIDs1(5))
Simulink.sdi.deleteSignal(signalIDs1(8))

Simulink.sdi.deleteSignal(signalIDs2(1))
Simulink.sdi.deleteSignal(signalIDs2(3))
Simulink.sdi.deleteSignal(signalIDs2(5))
Simulink.sdi.deleteSignal(signalIDs2(8))

**Compare Truncated Runs**

You can use the truncated runs you created with Simulink.sdi.copyRun and Simulink.sdi.deleteSignal to perform a comparison of the system's response to different types of failures. Then, open the Simulation Data Inspector to view the comparison results.

truncRunDiff = Simulink.sdi.compareRuns(truncRun1, truncRun2);

Simulink.sdi.view

**Input Arguments**

**runID** — Numeric run identifier
 scalar

Run ID for the run you want to copy. The Simulation Data Inspector assigns run IDs when it creates runs. You can get the run ID for your run using Simulink.sdi.getAllRunIDs or Simulink.sdi.getRunIDByIndex.

**Output Arguments**

**newRunID** — Run ID for the copy
 scalar

Run ID for the copy of the run.
runIndex — Run index for the copy
scalar

Index of the copy in the Simulation Data Inspector repository.

signalIDs — Numeric identifiers for the signals in the copy
matrix

Matrix containing the signal IDs for the copies of signals created in the copy of the run.

See Also
Simulink.sdi.createRun | Simulink.sdi.deleteRun | Simulink.sdi.deleteSignal

Topics
“Inspect and Compare Data Programmatically”

Introduced in R2011b
Simulink.sdi.copyRunViewSettings

Copy line style and color for signals from one run to another

**Syntax**

\[ \text{sigIDs} = \text{Simulink.sdi.copyRunViewSettings}(\text{run1}, \text{run2}, \text{plot}) \]

**Description**

\[ \text{sigIDs} = \text{Simulink.sdi.copyRunViewSettings}(\text{run1}, \text{run2}, \text{plot}) \] copies the line style and color specifications from \( \text{runID1} \) to \( \text{runID2} \) for matched signals. You can specify \( \text{run1} \) and \( \text{run2} \) with their run ID or as a Simulink.sdi.Run object. If \( \text{plot} \) is specified as true, Simulink.sdi.copyRunViewSettings also changes signal parameters in both runs so that aligned signals that are plotted come from the \( \text{run2} \). The function returns an array of signal identifiers for the signals that the Simulation Data Inspector aligned between the two runs. To learn more about how the Simulation Data Inspector aligns signals between runs, see “How the Simulation Data Inspector Compares Data”.

**Examples**

**Copy View Settings to a Run**

This example shows how to copy view settings from one run to another and how to create figures using the Simulink.sdi.CustomSnapshot object.

**Simulate Your Model and Get a Run Object**

Configure the \( \text{vdp} \) model to save output data, and run a simulation to create data.

```matlab
load_system('vdp')
set_param('vdp','SaveFormat','Dataset','SaveOutput','on')
set_param('vdp/Mu','Gain','1');
sim('vdp');
```
Use the Simulation Data Inspector programmatic interface to access the run data.

```matlab
runIndex = Simulink.sdi.getRunCount;
runID = Simulink.sdi.getRunIDByIndex(runIndex);
vdpRun = Simulink.sdi.getRun(runID);
```

**Modify Signal View Settings**

Use the `Simulink.sdi.Run` object to access signals in the run. Then, modify the signal view settings. This example specifies the line color and style for each signal. The view settings for the run comprise the view settings for each signal and view settings specified for the plot area.

```matlab
sig1 = vdpRun.getSignalByIndex(1);
sig2 = vdpRun.getSignalByIndex(2);

sig1.LineColor = [0 0 1];
sig1.LineDashed = '-.';

sig2.LineColor = [1 0 0];
sig2.LineDashed = ':';
```

**Capture a Snapshot from the Simulation Data Inspector**

Create a `Simulink.sdi.CustomSnapshot` object and use the `Simulink.sdi.snapshot` function to programmatically capture a snapshot of the contents of the Simulation Data Inspector.

```matlab
snap = Simulink.sdi.CustomSnapshot;
```

You can use properties of the `Simulink.sdi.CustomSnapshot` object to configure the plot settings, like the subplot layout and axis limits, and to plot signals. When you use a `Simulink.sdi.CustomSnapshot` object to create your figure, these plot settings do not affect the Simulation Data Inspector.

```matlab
snap.Rows = 2;
snap.YRange = {{-2.25 2.25}, [-3 3]};
snap.plotOnSubPlot(1,1,sig1,true)
snap.plotOnSubPlot(2,1,sig2,true)
```

Use `Simulink.sdi.snapshot` to generate the figure you specified in the properties of the `Simulink.sdi.CustomSnapshot` object.

```matlab
fig = Simulink.sdi.snapshot("from","custom","to","figure","settings",snap);
```
Copy the View Settings to a New Simulation Run

Simulate the model again, with a different \( \mu \) value. Use the Simulation Data Inspector programmatic interface to access the simulation data.

```MATLAB
set_param('vdp/Mu','Gain','5')
sim('vdp');
runIndex2 = Simulink.sdi.getRunCount;
runID2 = Simulink.sdi.getRunIDByIndex(runIndex2);
run2 = Simulink.sdi.getRun(runID2);
```

To create a plot of the new output data that looks like the one you created in the previous step, you can copy the view settings to the run in a single line of code using
Simulink.sdi.copyRunViewSettings. The Simulink.sdi.copyRunViewSettings function does not automatically update plot settings in Simulink.sdi.CustomSnapshot objects, so specify the plot input as false.

\[
sigIDs = \text{Simulink.sdi.copyRunViewSettings}(\text{runID}, \text{runID2}, \text{false});
\]

**Capture a Snapshot of the New Simulation Run**

Use the Simulink.sdi.CustomSnapshot object to capture a snapshot of the new simulation run. First, clear the signals from the subplots. Then, plot the signals from the new run and capture another snapshot.

\[
\begin{align*}
\text{snap.clearSignals} \\
\text{snap.YRange} &= \{[-2.25 \ 2.25], [-8 \ 8]\}; \\
\text{snap.plotOnSubPlot}(1,1,\text{sigIDs}(1),\text{true}) \\
\text{snap.plotOnSubPlot}(2,1,\text{sigIDs}(2),\text{true}) \\
\text{fig} &= \text{snap.snapshot("to","figure"});
\end{align*}
\]
**Input Arguments**

**run1** — Simulation Data Inspector run ID for source run  
scalar | `'Simulink.sdi.Run'` object

Run with the view settings you want to copy specified with its run ID or `Simulink.sdi.Run` object. The Simulation Data Inspector assigns run IDs when it creates runs. You can get the run ID for your run using `Simulink.sdi.getAllRunIDs` or `Simulink.sdi.getRunIDByIndex`.

**run2** — Simulation Data Inspector run ID for destination run  
scalar | `'Simulink.sdi.Run'` object
Run you want to copy the view settings to, specified with its run ID or Simulink.sdi.Run object. The Simulation Data Inspector assigns run IDs when it creates runs. You can get the run ID for your run using Simulink.sdi.getAllRunIDs or Simulink.sdi.getRunIDByIndex.

plot — Specify whether to update plotted signals
true | false

Specifies whether the Simulation Data Inspector changes the plot settings in the runs corresponding to run1 and run2.

- When plot is true, the Simulation Data Inspector modifies the signal parameters so that the aligned signals that are plotted come from run2.
- When plot is false, the Simulation Data Inspector does not change which signals are plotted.

Data Types: logical

Output Arguments

sigIDs — Signal IDs for aligned signals
matrix

Matrix containing the signal IDs for signals in run2 that aligned with signals in run1 and had view settings modified.

See Also

Simulink.sdi.Run | Simulink.sdi.Signal | Simulink.sdi.copyRun | Simulink.sdi.setMarkersOn | Simulink.sdi.view

Topics

“Inspect and Compare Data Programmatically”
“Create Plots Using the Simulation Data Inspector”

Introduced in R2016a
Simulink.sdi.createRun

**Package:** Simulink.sdi

Create a run in the Simulation Data Inspector

**Syntax**

```
runID = Simulink.sdi.createRun
runID = Simulink.sdi.createRun(runName)
runID = Simulink.sdi.createRun(var)
runID = Simulink.sdi.createRun(runName,'vars',var,var2,...,varn)
runID = Simulink.sdi.createRun(runName,'namevalue',sourceNames, sigValues)
runID = Simulink.sdi.createRun(runName,'file',filename)
```

```
[runID,runIndex] = Simulink.sdi.createRun(____)
[runID,runIndex,signalIDs] = Simulink.sdi.createRun(____)
```

**Description**

`runID = Simulink.sdi.createRun` creates an empty, unnamed run in the Simulation Data Inspector repository and returns the run ID. You can use `Simulink.sdi.getRun` to get a `Simulink.sdi.Run` object for the run, which allows you to add metadata and signals to the run.

`runID = Simulink.sdi.createRun(runName)` creates an empty run named `runName`.

`runID = Simulink.sdi.createRun(var)` creates a run named `var` containing the data in the workspace variable `var`.

`runID = Simulink.sdi.createRun(runName,'vars',var,var2,...,varn)` creates a run named `runName` containing data from one or more variables in the base workspace. The signals in the run take their names from the variable names.
runID = Simulink.sdi.createRun(runName,'namevalue',sourceNames, sigValues) creates a run named runName with the data in the cell array sigValues. The cell array of sourceNames specifies names to use as the source for the sigValues data.

runID = Simulink.sdi.createRun(runName,'file',filename) creates a run with data from the MAT, CSV, MDF, or Microsoft Excel file specified by filename.

[runID,runIndex] = Simulink.sdi.createRun( ___ ) returns the run ID and run index for the run created in the Simulation Data Inspector repository.

[runID,runIndex,signalIDs] = Simulink.sdi.createRun( ___ ) returns the run ID, run index within the Simulation Data Inspector, and the signal IDs for the signals in the run.

Examples

Create a Run in the Simulation Data Inspector

This example shows several ways to create a run in the Simulation Data Inspector for your data. You can create a run from simulation outputs, workspace data, and from a file.

Create Data

You can create Simulation Data Inspector runs from workspace data or from a file. The namevalue syntax for Simulink.sdi.createRun can be useful for hierarchical data. Create example data to use in each scenario.

% Create timeseries workspace data
time = linspace(0,20,101);

sine_vals = sin(2*pi/5*time);
sine_ts = timeseries(sine_vals,time);
sine_ts.Name = 'Sine, T=5';

cos_vals = cos(2*pi/8*time);
cos_ts = timeseries(cos_vals,time);
cos_ts.Name = 'Cosine, T=8';

% Create Dataset workspace data
```matlab
sinusoids_ds = Simulink.SimulationData.Dataset;
sinusoids_ds = sinusoids_ds.add(cos_ts);
sinusoids_ds = sinusoids_ds.add(sine_ts);

doubSine = 2*sine_ts;
doubCos = 2*cos_ts;

doubSinusoids_ds = Simulink.SimulationData.Dataset;
doubSinusoids_ds = doubSinusoids_ds.add(doubSine);
doubSinusoids_ds = doubSinusoids_ds.add(doubCos);

% Save workspace data to a MAT-file
save sinusoids.mat sine_ts cos_ts

Create a Run with a Run Object

You can create a run object using the Simulink.sdi.Run object's create method and then add data to the run using the add method. The add method supports all loading and logging data formats.

% Create a run
sinusoidsRun = Simulink.sdi.Run.create;
sinusoidsRun.Name = 'Sinusoids';
sinusoidsRun.Description = 'Sine and cosine signals of different frequencies';

% Add timeseries data to run
sinusoidsRun.add('vars',sine_ts,cos_ts);

Create a Run for a Variable

When you have only one signal of interest that you want to inspect and compare in the Simulation Data Inspector you can use a simple syntax to create the run. The run takes the same name as the variable.

runID = Simulink.sdi.createRun(sine_ts);

Create a Named Empty Run

With this syntax, you can create an empty run and specify its name. Then, you can use the returned runID to add data to the run.

runID = Simulink.sdi.createRun('My Waves');
signalID = Simulink.sdi.addToRun(runID,'vars',cos_ts);
```
Create a Run from Multiple Workspace Variables

When your signals of interest are in multiple variables in your workspace, use the 'vars' syntax. You can also use this syntax to provide a custom name for a run created from a single variable.

```matlab
runID = Simulink.sdi.createRun('My Sinusoids','vars',sine_ts,cos_ts);
```

Create a Run and Specify the Source for the Data

Providing a name for the source of the run data can be helpful, particularly with hierarchical data. Use the 'namevalue' syntax to specify data source names.

```matlab
runID = Simulink.sdi.createRun('Waves','namevalue',{'Sinusoids','BigSinusoids'},[sinusoids_ds,doubSinusoids_ds]);
```

Create a Run from Data in a File

You can create a run in the Simulation Data Inspector directly from a file of data using the 'file' syntax.

```matlab
runID = Simulink.sdi.createRun('Wave Data','file','sinusoids.mat');
```

View Runs in the Simulation Data Inspector

Look at the runs in the Inspect pane to see the results from each run creation method. You can select a run or a signal to view its metadata.

```matlab
Simulink.sdi.view
```

Create a Simulation Data Inspector Run and Access Signal Data

This example shows how to access signal data when you create a run in the Simulation Data Inspector.

Generate Data for Run

For this example, create timeseries data for sine and cosine signals.

```matlab
% Create timeseries workspace data
time = linspace(0, 20, 101);
sine_vals = sin(2*pi/5*time);
```
sine_ts = timeseries(sine_vals,time);
sine_ts.Name = 'Sine, T=5';

cos_vals = cos(2*pi/8*time);
cos_ts = timeseries(cos_vals,time);
cos_ts.Name = 'Cosine, T=8';

Create a Run and Return Signal IDs

You can use the Simulink.sdi.createRun syntax with multiple return arguments to get the signal IDs more directly instead of accessing the signal IDs through a Simulink.sdi.Run object.

[runID,runIndex,sigIDs] = Simulink.sdi.createRun('Sinusoids','vars',
    sine_ts,cos_ts);

cosID = sigIDs(2);
cosSig = Simulink.sdi.getSignal(cosID);

Modify Signal Properties and View in the Simulation Data Inspector

You can use the Simulink.sdi.Signal object to view and modify signal properties and to plot signals in the Simulation Data Inspector.

cosSig.Checked = true;
cosSig.AbsTol = 0.05;
Simulink.sdi.view

Input Arguments

runName — Name for the run
character vector

Name for the run in the Simulation Data Inspector.

Example: 'Baseline Simulation'

var — Variable name
variable

Variable in the base workspace containing the data you want to create a run for in the Simulation Data Inspector. The Simulation Data Inspector requires an associated time
vector for your data. Simulink.sdi.createRun supports all loading and logging data formats, including timeseries and Simulink.SimulationData.Dataset.

Provide one var argument when var is the only argument and one or more var arguments when you specify 'vars' for the second argument.

Example: myData

**sourceNames — Cell array of signal names**
cell array

Cell array of character vectors in which each element is the name of the source for data in the run. Provide a sigNames input when you specify 'namevalue' for the second argument.

Example: {'sig1','sig2'}

**sigValues — Signal data**
cell array

Cell array of variables containing signal data for the run. Provide a sigValues input when you specify 'namevalue' for the second argument.

Example: {var1,var2}

**filename — Name of file containing data for run**
character vector

File name of the file containing your run data. Provide a filename input when you specify 'file' for the second argument. You can create a run from a MAT, CSV, or Microsoft Excel file. You can also create a run from an MDF-file with an mdf, mf4, mf3, data, or dat file extension.

Example: 'simulation.mat'

### Output Arguments

**runID — Run identifier**
scalar

Run identifier for the new run.
runIndex — Run index
scalar

Index of the new run in the Simulation Data Inspector repository.

signalIDs — Signal IDs for signals in run
vector

Vector containing the signal IDs for the signals in the run.

See Also
Simulink.sdi.Run | Simulink.sdi.addToRun | Simulink.sdi.copyRun |
Simulink.sdi.createRunOrAddToStreamedRun | Simulink.sdi.deleteRun |
Simulink.sdi.getRun

Topics
“Inspect and Compare Data Programmatically”
“View Data with the Simulation Data Inspector”

Introduced in R2011b
Simulink.sdi.createRunOrAddToStreamedRun

**Package:** Simulink.sdi

Create a single run for all simulation outputs

**Syntax**

```matlab
runID = Simulink.sdi.createRunOrAddToStreamedRun(mdl,runName,varSources,varValues)
```

**Description**

`runID = Simulink.sdi.createRunOrAddToStreamedRun(mdl,runName,varSources,varValues)` creates a run with the data `varValues` if no run exists in the Simulation Data Inspector repository for the model `mdl`. If one or more runs for the model `mdl` exist in the Simulation Data Inspector repository, the function adds `varValues` to the most recent run associated with `mdl`. The run is named according to `runName`, and the sources for the data in `varValues` are named according to `varSources`.

**Examples**

**Add Signals to a Run**

This example shows how to use `Simulink.sdi.createRunOrAddToStreamedRun` to add data to an existing run for a model. In this example, you add logged states data to the run created through simulation.

**Simulate the Model**

Simulate the model to generate data. The model `slexAircraftExample` is configured to log outputs, so the Simulation Data Inspector automatically creates a run with the logged outport data. Using this simulation syntax, `out` contains the outport data (`yout`) and the states data (`xout`).
load_system('slexAircraftExample')
out = sim('slexAircraftExample','ReturnWorkspaceOutputs','on',...
'SaveFormat','Dataset');

**Add Logged States Data to Run**

The Simulation Data Inspector automatically created a run for the logged outport data. Add the logged state data to the existing run using `Simulink.sdi.createRunOrAddToStreamedRun`.

```matlab
Simulink.sdi.createRunOrAddToStreamedRun('slexAircraftExample','Run 1',...
{'out'},{out});
```

**Open the Simulation Data Inspector to View Results**

Using `Simulink.sdi.createRunOrAddToStreamedRun` avoids redundancy in the data shown in the Simulation Data Inspector. Using `Simulink.sdi.createRun` to bring the states data into the Simulation Data Inspector creates a second run. `Simulink.sdi.addToRun` creates a duplicate signal from the outport data. Using `Simulink.sdi.createRunOrAddToStreamedRun`, you can include all simulation data in a single run without duplicating any signals.

```matlab
Simulink.sdi.view
```

**Input Arguments**

- `mdl` — Name of model that created simulation data  
  character vector

  Name of the model the simulation data is from, specified as a character vector.

  Example: `'sldemo_absbrake'`

- `runName` — Name for the run  
  character vector

  Name for the new or augmented run. If `Simulink.sdi.createRunOrAddToStreamedRun` adds data to an existing run, the run is renamed according to `runName`.

  Example: `'Run 1'`
varSources — Names to use for the sources of data
cell array of character vectors

Names for the sources of the data in varValues.
Example: {'sig1','sig2'}

varValues — Data to add to run
cell array

Cell array of data to incorporate into the run.
Simulink.sdi.createRunOrAddToStreamedRun supports data in all logging and loading formats, including timeseries and Simulink.SimulationData.Dataset.
Example: {sig1,sig2}

Output Arguments

runID — Run identifier
scalar

Run identifier for the new or augmented run.

See Also

Simulink.sdi.Run | Simulink.sdi.addToRun | Simulink.sdi.createRun |
Simulink.sdi.getAllRunIDs | Simulink.sdi.getRun |
Simulink.sdi.getRunCount | Simulink.sdi.isValidRunID |
Simulink.sdi.view

Topics

“Inspect and Compare Data Programmatically”
“View Data with the Simulation Data Inspector”

Introduced in R2017a
Simulink.sdi.deleteRun

Package: Simulink.sdi

Delete a run from the Simulation Data Inspector repository

Syntax

Simulink.sdi.deleteRun(runID)

Description

Simulink.sdi.deleteRun(runID) deletes the run corresponding to runID. When you delete a run, the indices of all runs following the deleted run change to account for the change in the run count. Deleting a run does not change any run IDs.

Examples

Delete a Run

You can delete a run from the Simulation Data Inspector repository to free up memory space or to declutter your workspace from data you do not need.

% Load and simulate sldemo_fuelsys model
load_system('sldemo_fuelsys')
sim('sldemo_fuelsys')

% Get the run ID for the run
runIDs = Simulink.sdi.getAllRunIDs;
runID = runIDs(end);
% Delete the run
Simulink.sdi.deleteRun(runID)

Input Arguments

runID — Run identifier
calar

Run ID for the run you want to delete. You can get the run ID for a run using Simulink.sdi.getAllRunIDs or Simulink.sdi.getRunIDByIndex.

See Also

Simulink.sdi.clear | Simulink.sdi.copyRun | Simulink.sdi.deleteSignal | Simulink.sdi.getAllRunIDs | Simulink.sdi.getArchiveRunLimit | Simulink.sdi.getRunIDByIndex | Simulink.sdi.setArchiveRunLimit

Topics

“Inspect and Compare Data Programmatically”

Introduced in R2011b
Simulink.sdi.deleteSignal

**Package:** Simulink.sdi

Delete a signal from the Simulation Data Inspector repository

**Syntax**

Simulink.sdi.deleteSignal(sigID)

**Description**

Simulink.sdi.deleteSignal(sigID) deletes the signal corresponding to sigID from the Simulation Data Inspector repository.

**Examples**

**Compare a Subset of Signals**

This example shows how to use Simulink.sdi.copyRun and Simulink.sdi.deleteSignal to create a copy of a run that contains a subset of the signals from the original run. You can use the copy to analyze and run comparisons on a subset of signals while still holding onto the original run that has all of the signals. For example, the model sldemo_fuelsys is configured to log ten signals. To compare the system's responses to different types of failures, you don't need to run the comparison on all of the logged signals. Deleting signals that do not represent the system's response before running the comparison saves processing time and simplifies the view of the results.

**Create Runs**

Load the model sldemo_fuelsys and run simulations to create runs in the Simulation Data Inspector. The first run simulates a failure of the throttle angle sensor, and the second run simulates a failure of the exhaust gas oxygen sensor.
load_system('sldemo_fuelsys')
modelWorkspace = get_param('sldemo_fuelsys','modelworkspace');
modelWorkspace.assignin('throttle_sw',0)
modelWorkspace.assignin('ego_sw',1)
sim('sldemo_fuelsys')
modelWorkspace.assignin('throttle_sw',1)
modelWorkspace.assignin('ego_sw',0)
sim('sldemo_fuelsys')

Copy the Run

Use the Simulation Data Inspector's programmatic interface to get Simulink.sdi.Run objects for the simulations, and then create copies of the runs.

% Get runs
runIDs = Simulink.sdi.getAllRunIDs;
runID1 = runIDs(end-1);
runID2 = runIDs(end);
run1 = Simulink.sdi.getRun(runID1);
run2 = Simulink.sdi.getRun(runID2);

% Create a copy of each run, truncRun
[truncRun1,runIndex1,signalIDs1] = Simulink.sdi.copyRun(runID1);
[truncRun2,runIndex2,signalIDs2] = Simulink.sdi.copyRun(runID2);

Delete Signals in Run Copy

The sldemo_fuelsys model is configured to log the values of the fault switches along with several signals representing the system’s response. When you compare the system’s response when a throttle angle sensor fails to its response when an exhaust gas oxygen sensor fails, comparing the fault switch states does not provide new information. Therefore, delete the switch signals before running the comparison to eliminate unnecessary computations.

Simulink.sdi.deleteSignal(signalIDs1(1))
Simulink.sdi.deleteSignal(signalIDs1(3))
Simulink.sdi.deleteSignal(signalIDs1(5))
Simulink.sdi.deleteSignal(signalIDs1(8))
Simulink.sdi.deleteSignal(signalIDs2(1))
Simulink.sdi.deleteSignal(signalIDs2(3))
Simulink.sdi.deleteSignal(signalIDs2(5))
Simulink.sdi.deleteSignal(signalIDs2(8))

Compare Truncated Runs

You can use the truncated runs you created with Simulink.sdi.copyRun and Simulink.sdi.deleteSignal to perform a comparison of the system's response to different types of failures. Then, open the Simulation Data Inspector to view the comparison results.

truncRunDiff = Simulink.sdi.compareRuns(truncRun1,truncRun2);
Simulink.sdi.view

Input Arguments

sigID — Signal ID
scalar

Unique number identifying the signal within the Simulation Data Inspector repository. You can get the signal ID for a signal as a return from Simulink.sdi.createRun or using the Simulink.sdi.Run object's methods.

See Also
Simulink.sdi.Run | Simulink.sdi.Signal | Simulink.sdi.copyRun | Simulink.sdi.createRun

Topics
“Inspect and Compare Data Programmatically”
“Organize Your Simulation Data Inspector Workspace”

Introduced in R2016a
Simulink.sdi.enablePCTSupport

Control how the Simulation Data Inspector works with the Parallel Computing Toolbox

Syntax

Simulink.sdi.enablePCTSupport(mode)

Description

Simulink.sdi.enablePCTSupport(mode) enables support for automatic data import from parallel workers into the Simulation Data Inspector, according to the mode specified by mode. You can configure the Simulation Data Inspector to import no worker data, only data from local workers, or data from all workers — local and remote. You can also configure the parallel worker support as manual, where you manually select runs to import to the Simulation Data Inspector using the Simulink.sdi.sendWorkerRunToClient function. By default, the Simulation Data Inspector automatically imports runs from local workers.

Examples

Enable Parallel Support for All Workers

Configure Simulation Data Inspector parallel worker support to import the output automatically from both local and remote workers.

Simulink.sdi.enablePCTSupport('all')

Disable Support for Parallel Workers

To prevent the output from any Parallel Computing Toolbox workers from automatically importing to the Simulation Data Inspector, disable Parallel Computing Toolbox support.
Manually Send Runs from Parallel Workers to the Simulation Data Inspector

This example shows how to use Simulink.sdi.sendWorkerRunToClient to send runs created using parallel workers manually to the Simulation Data Inspector.

Setup

This example runs several simulations of the vdp model, varying the value of the gain, Mu. To set up for the parallel simulation, define a vector of Mu values and configure the Simulation Data Inspector for manual Parallel Computing Toolbox support.

```matlab
% Enable manual Parallel Computing Toolbox support
Simulink.sdi.enablePCTSupport('manual');

% Choose several Mu values
MuVals = [1 2 3 4];
```

Initialize Parallel Workers

Use parpool to start a pool of four parallel workers. This example calls parpool inside an if statement so you only create a parallel pool if you don't already have one. You can use spmd to run initialization code common to all workers. For example, load the vdp model and select signals to log to runs that we can send to the Simulation Data Inspector on the client MATLAB. To avoid data concurrency issues when simulating with sim in parfor, create a temporary directory on each worker. After the simulations complete, another spmd block deletes the temporary directories.

```matlab
p = gcp('nocreate');
if isempty(p)
    parpool(4);
end
```

Starting parallel pool (parpool) using the 'local' profile ... connected to 4 workers.

```matlab
spmd
```
% Load system and select signals to log
load_system('vdp')
Simulink.sdi.markSignalForStreaming('vdp/x1', 1, 'on')
Simulink.sdi.markSignalForStreaming('vdp/x2', 1, 'on')

% Create temporary directory for simulation on worker
workDir = pwd;
addpath(workDir)
tempDir = tempname;
mkdir(tempDir)
set(workDir, 'ParentFolder', tempDir)
cd(tempDir)

end

Run Parallel Simulations with parfor

To stream data from parallel workers to the Simulation Data Inspector, you have to run parallel simulations using parfor. Each worker runs a vdp simulation with a different value of Mu. Simulink cannot access the contents of the parfor loop, so the variable MuVal is defined in the worker's workspace, where the vdp model can see it, using assignin.

parfor (index = 1:4)
    % Set value of Mu in the worker's base workspace
    assignin('base', 'MuVal', MuVals(index));
    % Modify the value of Mu in the model and simulate
    set_param('vdp/Mu', 'Gain', 'MuVal')
    sim('vdp')
end

Access Data and Send Run to Client MATLAB

You can use the Simulation Data Inspector programmatic interface on the worker the same way you would in the client MATLAB. This example creates a Simulink.sdi.Run object and attaches the value of Mu used in the simulation with the Tag property.

% Attach metadata to the run
IDs = Simulink.sdi.getAllRunIDs;
lastIndex = length(IDs);
runID = Simulink.sdi.getRunIDByIndex(lastIndex);
parRun = Simulink.sdi.getRun(runID);
parRun.Tag = strcat('Mu = ', num2str(MuVals(index)));

% Send the run to the Simulation Data Inspector on the client MATLAB
Simulink.sdi.sendWorkerRunToClient

end

Close Temporary Directories and View Runs in the Simulation Data Inspector

Use another `spmd` section to delete the temporary directories created on the workers once the simulations complete. In each simulation, `Simulink.sdi.sendWorkerRunToClient` imported runs from all the workers into the Simulation Data Inspector. You can view the data and check the run properties to see the value of `Mu` used during simulation.

```matlab
spmd
    % Remove temporary directories
    cd(workDir)
    rmdir(tempDir, 's')
    rmpath(workDir)
end

Simulink.sdi.view
```

Access Data from a Parallel Simulation

This example executes parallel simulations of the model `slexAircraftExample` with different input filter time constants and shows several ways to access the data using the Simulation Data Inspector programmatic interface.

Setup

Start by ensuring the Simulation Data Inspector is empty and Parallel Computing Toolbox support is configured to import runs created on local workers automatically. Then, create a vector of filter parameter values to use in each simulation.

```matlab
% Make sure the Simulation Data Inspector is empty, and PCT support is % enabled.
Simulink.sdi.clear
Simulink.sdi.enablePCTSupport('local')
```
% Define Ts values
Ts_vals = [0.01, 0.02, 0.05, 0.1, 0.2, 0.5, 1];

Initialize Parallel Workers

Use gcp to create a pool of local workers to run parallel simulations if you don't already have one. In an spmd code block, load the slexAircraftExample model and select signals to log. To avoid data concurrency issues using sim in parfor, create a temporary directory for each worker to use during simulations.

p = gcp;
Starting parallel pool (parpool) using the 'local' profile ...
connected to 4 workers.

spmd

% Load system and select signals to log
load_system('slexAircraftExample')
Simulink.sdi.markSignalForStreaming('slexAircraftExample/Pilot', 1, 'on')
Simulink.sdi.markSignalForStreaming('slexAircraftExample/Aircraft Dynamics Model', 4, 'on')

% Create temporary directory on each worker
workDir = pwd;
addpath(workDir)
tempDir = tempname;
mkdir(tempDir)
cd(tempDir)
end

Run Parallel Simulations

Use parfor to run the seven simulations in parallel. Select the value for Ts for each simulation, and modify the value of Ts in the model workspace. Then, run the simulation and build an array of Simulink.sdi.WorkerRun objects to access the data with the Simulation Data Inspector. After the parfor loop, use another spmd segment to remove the temporary directories from the workers.

parfor index = 1:7

% Select value for Ts
Ts_val = Ts_vals(index);
% Change the filter time constant and simulate
modelWorkspace = get_param('slexAircraftExample','modelworkspace');
modelWorkspace.assignin('Ts',Ts_val)
sim('slexAircraftExample')

% Create a worker run for each simulation
workerRun(index) = Simulink.sdi.WorkerRun.getLatest

end

spmd

% Remove temporary directories
cd(workDir)
rmdir(tempDir, 's')
rmpath(workDir)
end

Get Dataset Objects from Parallel Simulation Output

The `getDataset` method puts the data from a `WorkerRun` into a `Dataset` object so you can easily post-process.

ds(7) = Simulink.SimulationData.Dataset;

for a = 1:7
    ds(a) = workerRun(a).getDataset;
end
ds(1)

ans =
Simulink.SimulationData.Dataset ' ' with 2 elements

<table>
<thead>
<tr>
<th>Name</th>
<th>BlockPath</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>[1x1 Signal] alpha, rad ...rcraftExample/Aircraft Dynamics Model</td>
</tr>
<tr>
<td>2</td>
<td>[1x1 Signal] Stick slexAircraftExample/Pilot</td>
</tr>
</tbody>
</table>

- Use braces `{ }` to access, modify, or add elements using index.
Get DatasetRef Objects from Parallel Simulation Output

For big data workflows, use the getDatasetRef method to reference the data associated with the WorkerRun.

```matlab
for b = 1:7
    datasetRef(b) = workerRun(b).getDatasetRef;
end
```

datasetRef(1)

```matlab
ans =
    DatasetRef with properties:
        Name: 'Run 3: slexAircraftExample'
        Run: [1×1 Simulink.sdi.Run]
    numElements: 2
```

Process Parallel Simulation Data in the Simulation Data Inspector

You can also create local Run objects to analyze and visualize your data using the Simulation Data Inspector API. This example adds a tag indicating the filter time constant value for each run.

```matlab
for c = 1:7
    Runs(c) = workerRun(c).getLocalRun;
    Ts_val_str = num2str(Ts_vals(c));
    desc = strcat('Ts = ', Ts_val_str);
    Runs(c).Description = desc;
    Runs(c).Name = strcat('slexAircraftExample run Ts=', Ts_val_str);
end
```

Clean Up Worker Repositories

Clean up the files used by the workers to free up disk space for other simulations you want to run on your worker pool.
Input Arguments

mode — Parallel worker support mode

'mode' — Parallel worker support mode

'local' (default) | 'none' | 'all' | 'manual'

Mode of Simulation Data Inspector support for importing runs from parallel workers.

- 'local' — The default behavior configures automatic import for runs generated on local workers.
- 'none' — Disables parallel worker support. No runs from local or remote workers import to the Simulation Data Inspector.
- 'all' — Enables automatic import for runs created from local and remote workers.
- 'manual' — Configures support for manual import of runs created by parallel workers using the Simulink.sdi.sendWorkerRunToClient function.

Data Types: char | string

See Also

Simulink.sdi.WorkerRun | Simulink.sdi.isPCTSupportEnabled | Simulink.sdi.sendWorkerRunToClient

Topics

“Inspect and Compare Data Programmatically”

Introduced in R2017b
Simulink.sdi.exportRun

Export Simulation Data Inspector run data to the workspace or a file

Syntax

dataset = Simulink.sdi.exportRun(runID)
Simulink.sdi.exportRun(runID,Name,Value)

Description

dataset = Simulink.sdi.exportRun(runID) creates a
Simulink.SimulationData.Dataset object in the base workspace with the data in
the Simulation Data Inspector run identified by runID.

Simulink.sdi.exportRun(runID,Name,Value) exports the data in the run
corresponding to runID to the base workspace or a file according to the options specified
by one or more name-value pair arguments. You can export data for one or more runs to a
MAT, MLDATX, or Microsoft Excel file. To export data for multiple runs to a file, you can
specify the runIDs input as a vector of run IDs.

Examples

Export Run Data

This example shows how to export data from a run in the Simulation Data Inspector to a
Simulink.SimulationData.Dataset object in the base workspace you can use to
further process your data. The method you choose to export your run depends on the
processing you do in your script. If you have a run object for the run, you can use the
export method to create a Simulink.SimulationData.Dataset object with the run
data in the base workspace. If you do not have a run object, use the
Simulink.sdi.exportRun function to export the run to the workspace.
Export Run Using Simulink.sdi.exportRun

Use the Simulink.sdi.export function to export run data to the workspace or a file when your workflow does not include creating a run object.

To create a run of simulation data, open the vdp model, mark signals for logging, and run a simulation.

```matlab
load_system('vdp')
SignalHandles = get_param('vdp', 'Lines');
Simulink.sdi.markSignalForStreaming(SignalHandles(5).Handle, 'on')
Simulink.sdi.markSignalForStreaming(SignalHandles(6).Handle, 'on')
out = sim('vdp');
```

Use the Simulink.sdi.getAllRunIDs function to access the most recently created run.

```matlab
runIDs = Simulink.sdi.getAllRunIDs;
runID = runIDs(end);
```

Use the Simulink.sdi.exportRun function to export the run data to a Dataset object in the workspace.

```matlab
Simulink.sdi.exportRun(runID);
```

Export Run Using export Method

When your workflow involves creating a Run object, you can use the export function to create a Simulink.SimulationData.Dataset object in the base workspace for further processing of the run data. For example, you need to access Run objects for simulation runs with signal data you want to compare using the Simulink.sdi.compareSignals function.

Load a model and mark signals for logging. Then, simulate the model to create run data.

```matlab
load_system('vdp')
SignalHandles = get_param('vdp', 'Lines');
Simulink.sdi.markSignalForStreaming(SignalHandles(5).Handle, 'on')
Simulink.sdi.markSignalForStreaming(SignalHandles(6).Handle, 'on')
```
sim('vdp');

Use the Simulink.sdi.getAllRunIDs function to access the run ID for the most recently created run. Then, use the Simulink.sdi.getRun function to access the Run object corresponding to the run.

runIDs = Simulink.sdi.getAllRunIDs;
runID = runIDs(end);
vdpRun = Simulink.sdi.getRun(runID);

Use the export function to export the run data to a Dataset object in the workspace.

simDataset = export(vdpRun);

### Input Arguments

**runID — Run identifier**

Scalar | Vector

Run identifier for the run you want to export to the workspace or a file. When you export data to a file, you can export more than one run by specifying the runID input as a vector of Simulation Data Inspector run IDs.

The Simulation Data Inspector assigns a unique run ID to each run. You can get the run ID for one or more runs using Simulink.sdi.getAllRunIDs and Simulink.sdi.getRunIDByIndex.

### Name-Value Pair Arguments

Specify optional comma-separated pairs of Name, Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside quotes. You can specify several name and value pair arguments in any order as Name1, Value1, ..., NameN, ValueN.

Example: 'to', 'file'

**to — Specify where to export data**

'variable' (default) | 'file'

Specify whether to export data to a variable in the base workspace or to a file.
When you export data to a file, you must also specify a file name using the 'filename' name-value pair argument. You can specify a file name with a MAT, MLDATX, or XLSX extension.

When you export a run to a MAT-file, the data is saved in a Simulink.SimulationData.Dataset object. When you export more than one run to a MAT-file using the Simulink.sdi.exportRun function, the data is saved in a Dataset object, where each element is a Dataset object containing the data for an exported run.

Data exported to a Microsoft Excel file is saved with the format described in “Simulation Data Inspector Import and Export File Format”.

When you export data to a Microsoft Excel file, you can specify additional options using the 'overwrite', 'metadata', and 'sharetimecolumn' name-value pairs.

Example: 'to','file'

filename — Name of file to contain exported data

string

Name of the file to contain the exported data, specified as a string or character array. Include a MAT, MLDATX, or XLSX extension in the file name to specify whether to export to a MAT-file, MLDATX-file, or a Microsoft Excel file. When you do not specify an extension with the file name, the data exports to a MAT-file.

Use the 'filename' name-value pair argument when you specify the 'to' name-value pair argument with the value 'file'.

When you export data to a Microsoft Excel file, you can specify additional options using the 'overwrite', 'metadata', and 'sharetimecolumn' name-value pair arguments.

Example: 'filename','mySpreadsheet.xlsx'

overwrite — Specify data to overwrite in existing Microsoft Excel file

'field' (default) | 'sheetsonly'

Specify data to overwrite in existing Microsoft Excel file. When you export data to a Microsoft Excel file that already exists, you can control whether the export overwrites the entire file or only the affected sheets in the file.

- 'file' — Overwrite the entire file with the exported data.
- 'sheetsonly' — Only overwrite sheets of the Microsoft Excel file with data that corresponds to the exported data.
Example: 'overwrite','sheetsonly'

`metadata — Specify metadata to include in exported Microsoft Excel file [] (default) | string array`

Metadata to include in the exported Microsoft Excel file, specified in a string array. By default, the export operation does not include any metadata. You can export this metadata to the Microsoft Excel file:

- `dataType` — Signal data type
- `units` — Signal units
- `blockPath` — Path to the source block for logged signals
- `interp` — Signal interpolation method
- `portIndex` — Index of the port on the source block for logged signals

You can specify the desired metadata in any order you choose in the string array. The order of the metadata does not affect the format in the exported file, which always matches the description in “Simulation Data Inspector Import and Export File Format”.

Example: 'metadata',["units","dataType"]

`sharetimecolumn — Specify whether signals share time columns in exported Microsoft Excel file 'on' (default) | 'off'`

Specify whether to share time columns in the exported Microsoft Excel file for signals that have identical time data. By default, signals with the same time data share a time column in the exported file. When you specify the value as 'off', each signal in the exported file has its own time column.

Example: 'sharetimecolumn','off'

**Output Arguments**

`dataset — Dataset containing run data`  
*Simulink.SimulationData.Dataset*

`Simulink.SimulationData.Dataset` object containing the data from the run identified by `runID`. 

2-875
See Also
Simulink.SimulationData.Dataset | Simulink.sdi.Run |
Simulink.sdi.getAllRunIDs | Simulink.sdi.getRunIDByIndex |
Simulink.sdi.save

Topics
“Inspect and Compare Data Programmatically”
“Save and Share Simulation Data Inspector Data and Views”

Introduced in R2017a
Simulink.sdi.getAllRunIDs

**Package:** Simulink.sdi

Get all Simulation Data Inspector run identifiers

**Syntax**

```matlab
runIDs = Simulink.sdi.getAllRunIDs
```

**Description**

`runIDs = Simulink.sdi.getAllRunIDs` returns a matrix of the run identifiers for all runs in the Simulation Data Inspector repository.

**Examples**

**Get Run ID for a Simulation**

Many workflows that use the Simulation Data Inspector programmatic interface start with obtaining the ID for a simulation run. This example shows two different methods to use the programmatic interface to get the run ID for a run. You can use the run ID to create a `Simulink.sdi.Run` object to access run data and metadata, or you can use the run ID for a comparison.

**Simulate a Model to Create a Run**

The model `sldemo_fuelsys` is already configured for logging. When you simulate the model, the Simulation Data Inspector automatically creates a run and assigns it a run ID.

```matlab
% Load and simulate system
load_system('sldemo_fuelsys')
sim('sldemo_fuelsys')
```
Get Run ID Using Simulink.sdi.getAllRunIDs

Simulink.sdi.getAllRunIDs returns an array of all run IDs for runs in the Simulation Data Inspector repository in order, with the most recently created run at the end.

```matlab
% Get runID for most recent run
runIDs = Simulink.sdi.getAllRunIDs;
runID = runIDs(end);
```

Get Run ID Using Simulink.sdi.getRunIDByIndex

You can also use Simulink.sdi.getRunCount and Simulink.sdi.getRunIDByIndex to get the run ID for a run. This method is useful if you also want to use count as a counting variable to index through the runs in the Simulation Data Inspector repository.

```matlab
count = Simulink.sdi.getRunCount;
runID = Simulink.sdi.getRunIDByIndex(count);
```

Analyze Simulation Data with Signal Tolerances

You can change tolerance values on a signal-by-signal basis to evaluate the effect of a model parameter change. This example uses the slexAircraftExample model and the Simulation Data Inspector to evaluate the effect of changing the time constant for the low-pass filter following the control input.

Setup

Load the model, and mark the q, rad/sec and alpha, rad signals for logging. Then, simulate the model to create the baseline run.

```matlab
% Load example model
load_system('slexAircraftExample')

% Mark the q, rad/sec and alpha, rad signals for logging
Simulink.sdi.markSignalForStreaming('slexAircraftExample/Aircraft Dynamics Model',3,'on')
Simulink.sdi.markSignalForStreaming('slexAircraftExample/Aircraft Dynamics Model',4,'on')

% Simulate system
out_1 = sim('slexAircraftExample');
```

Modify Model Parameter

Modify the model parameter Ts in the model workspace to change the time constant of the input low-pass filter.
% Change input filter time constant
modelWorkspace = get_param('slexAircraftExample','modelworkspace');
assignin(modelWorkspace,'Ts',1)

% Simulate again
out_2 = sim('slexAircraftExample');

**Compare Runs and Inspect Results**

Use the Simulink.sdi.compareRuns function to compare the data from the simulations. Then, inspect the match property of the signal result to see whether the signals fell within the default tolerance of 0.

% Get run data
runIDs = Simulink.sdi.getAllRunIDs;
runID1 = runIDs(end - 1);
runID2 = runIDs(end);

% Compare runs
diffRun1 = Simulink.sdi.compareRuns(runID1,runID2);

% Get signal result
sig1Result1 = getResultByIndex(diffRun1,1);
sig2Result1 = getResultByIndex(diffRun1,2);

% Check whether signals matched
sig1Result1.Status
ans = OutOfTolerance

sig2Result1.Status
ans = OutOfTolerance

**Compare Runs with Signal Tolerances**

The signals did not match within the default tolerance of 0. To further analyze the effect of the time constant change, add signal tolerances to the comparison with the baseline signal properties to determine the tolerance required for a pass. This example uses a combination of time and absolute tolerances.

% Get signal object for sigID1
run1 = Simulink.sdi.getRun(runID1);
sigID1 = getSignalIDByIndex(run1,1);
sigID2 = getSignalIDByIndex(run1,2);

sig1 = Simulink.sdi.getSignal(sigID1);
sig2 = Simulink.sdi.getSignal(sigID2);

% Set tolerances for q, rad/sec
sig1.AbsTol = 0.1;
sig1.TimeTol = 0.6;

% Set tolerances for alpha, rad
sig2.AbsTol = 0.2;
sig2.TimeTol = 0.8;

% Run the comparison again
diffRun2 = Simulink.sdi.compareRuns(runID1,runID2);
sig1Result2 = getResultByIndex(diffRun2,1);
sig2Result2 = getResultByIndex(diffRun2,2);

% Check the result
sig1Result2.Status
ans = WithinTolerance

sig2Result2.Status
ans = WithinTolerance

Output Arguments

runIDs — Matrix of Simulation Data Inspector run IDs
matrix

Matrix of run IDs in the Simulation Data Inspector repository.

See Also
Simulink.sdi.Run | Simulink.sdi.compareRuns | Simulink.sdi.copyRun | Simulink.sdi.copyRunViewSettings | Simulink.sdi.deleteRun |
Simulink.sdi.exportRun | Simulink.sdi.getRun | Simulink.sdi.getRunCount | Simulink.sdi.isValidRunID

**Topics**

“Inspect and Compare Data Programmatically”

**Introduced in R2017a**
Simulink.sdi.getArchiveRunLimit

Package: Simulink.sdi

Determine configured run limit for Simulation Data Inspector archive

Syntax

limit = Simulink.sdi.getArchiveRunLimit

Description

limit = Simulink.sdi.getArchiveRunLimit returns the limit configured for the number of runs the Simulation Data Inspector stores in the archive. When the number of runs in the archive reaches the limit, the Simulation Data Inspector starts to delete runs from the archive on a first-in, first-out basis. When another run enters the archive, the Simulation Data Inspector deletes the run that has been in the archive the longest. A limit of -1 indicates the archive has no limit for the number of runs it stores. A limit of 0 means that the archive cannot contain any runs.

Tip To retain data for only the current run, configure the Simulation Data Inspector to automatically archive runs and set the archive run limit to 0.

Examples

Configure the Simulation Data Inspector Archive

You can configure the Simulation Data Inspector to retain only the logged data for your current simulation. In iterative design and debugging workflows, this configuration helps prevent accumulation of unwanted logged data on disk. First, check the configuration of the Simulation Data Inspector archive. You can save the parameters to restore your preferences after you finish designing or debugging.
limit = Simulink.sdi.getArchiveRunLimit;
mode = Simulink.sdi.getAutoArchiveMode;

Set the archive limit to 0 and configure the Simulation Data Inspector to automatically archive simulation runs.

Simulink.sdi.setArchiveRunLimit(0)
Simulink.sdi.setAutoArchiveMode(true)

When you simulate your model, the Simulation Data Inspector deletes the previous run and updates the view to show signals in the current simulation.

When you finish designing or debugging your model, you can restore the Simulation Data Inspector archive back to its previous configuration.

Simulink.sdi.setArchiveRunLimit(limit)
Simulink.sdi.setAutoArchiveMode(mode)

Output Arguments

**limit** — Maximum number of runs to store in the Simulation Data Inspector archive  
scalar

Limit for the number of runs stored in the Simulation Data Inspector archive. When the number of runs in the archive reaches the limit, the Simulation Data Inspector starts to delete runs from the archive, on a first-in, first-out basis. A limit of -1 indicates there is no limit on the runs stored in the archive. A limit of 0 indicates that the archive cannot contain any runs.

See Also

Simulink.sdi.getAutoArchiveMode | Simulink.sdi.setArchiveRunLimit | Simulink.sdi.setAutoArchiveMode

Topics

“Iterate Model Design Using the Simulation Data Inspector”
Introduced in R2018b
Simulink.sdi.getAutoArchiveMode

**Package:** Simulink.sdi

Determine if the Simulation Data Inspector is configured to automatically archive

**Syntax**

mode = Simulink.sdi.getAutoArchiveMode

**Description**

mode = Simulink.sdi.getAutoArchiveMode returns a logical value that indicates whether the Simulation Data Inspector is configured to automatically archive simulation runs. When mode is true, the Simulation Data Inspector automatically archives simulation runs. When mode is false, the Simulation Data Inspector does not automatically archive.

**Examples**

**Configure the Simulation Data Inspector Archive**

You can configure the Simulation Data Inspector to retain only the logged data for your current simulation. In iterative design and debugging workflows, this configuration helps prevent accumulation of unwanted logged data on disk. First, check the configuration of the Simulation Data Inspector archive. You can save the parameters to restore your preferences after you finish designing or debugging.

```matlab
limit = Simulink.sdi.getArchiveRunLimit;
mode = Simulink.sdi.getAutoArchiveMode;
```

Set the archive limit to 0 and configure the Simulation Data Inspector to automatically archive simulation runs.
Simulink.sdi.setArchiveRunLimit(0)
Simulink.sdi.setAutoArchiveMode(true)

When you simulate your model, the Simulation Data Inspector deletes the previous run and updates the view to show signals in the current simulation.

When you finish designing or debugging your model, you can restore the Simulation Data Inspector archive back to its previous configuration.

Simulink.sdi.setArchiveRunLimit(limit)
Simulink.sdi.setAutoArchiveMode(mode)

**Output Arguments**

**mode** — Logical indication of Simulation Data Inspector automatic archive configuration

logical

Logical value that indicates whether the Simulation Data Inspector is configured to automatically archive simulation runs.

- **true** — The Simulation Data Inspector automatically archives simulation runs.
- **false** — The Simulation Data Inspector does not automatically archive simulation runs.

**See Also**

Simulink.sdi.getArchiveRunLimit | Simulink.sdi.setArchiveRunLimit | Simulink.sdi.setAutoArchiveMode

**Topics**

“Iterate Model Design Using the Simulation Data Inspector”

**Introduced in R2018b**
Simulink.sdi.getBorderOn

Get border display setting for time plots

Syntax

border = Simulink.sdi.getBorderOn

Description

border = Simulink.sdi.getBorderOn returns the current setting for whether borders are displayed on time plots in the Simulation Data Inspector. The return value corresponds to the setting for **Show border** in the Simulation Data Inspector **Time Plot Settings**. The setting applies to all time plots in the layout. By default, the Simulation Data Inspector displays the border on time plots.

Examples

Configure Time Plot Border Display

You can use the `Simulink.sdi.setBorderOn` function to show or hide the border on time plots in the Simulation Data Inspector. By default, the Simulation Data Inspector shows the border on time plots.

This example starts by showing how to use the `Simulink.sdi.getBorderOn` function to determine whether the border is currently displayed. Subsequent sections show the code to hide and show the border. To show the result, the example generates an image using the `Simulink.sdi.snapshot` function with settings specified by a `Simulink.sdi.CustomSnapshot` object.

```matlab
snapSettings = Simulink.sdi.CustomSnapshot;
snapSettings.Width = 300;
snapSettings.Height = 300;
```
Get Current Border Display Setting

Before modifying the border display setting for time plots, you can save the current setting to a variable in the workspace in case you want to restore the preference later.

```matlab
border = Simulink.sdi.getBorderOn;
```

Hide the Border on Time Plots

```matlab
Simulink.sdi.setBorderOn(false)
Simulink.sdi.snapshot('from','custom','settings',snapSettings);
```

Show the Border on Time Plots

```matlab
Simulink.sdi.setBorderOn(true)
Simulink.sdi.snapshot('from','custom','settings',snapSettings);
```
Output Arguments

**border — Border display setting for time plots**

```plaintext
type: true or 1 | false or 0
```

Border display setting for time plots in the Simulation Data Inspector.

- **true or 1** — Border displayed for time plots.
- **false or 0** — Border hidden for time plots.
See Also
Simulink.sdi.getGridOn | Simulink.sdi.getMarkersOn | Simulink.sdi.getTickLabelsDisplay | Simulink.sdi.getTicksPosition | Simulink.sdi.setBorderOn

Topics
“Create Plots Using the Simulation Data Inspector”

Introduced in R2019b
Simulink.sdi.getGridOn

Package: Simulink.sdi

Determine grid configuration for time plots

Syntax

grid = Simulink.sdi.getGridOn

Description

grid = Simulink.sdi.getGridOn returns the grid configuration for time plots in the Simulation Data Inspector. By default, the Simulation Data Inspector shows horizontal and vertical grid lines.

Examples

Configure Grid for Time Plots in the Simulation Data Inspector

You can use Simulink.sdi.setGridOn to configure the appearance of the grid for time plots in the Simulation Data Inspector. By default, the Simulation Data Inspector shows horizontal and vertical grid lines.

This example starts by showing how to use Simulink.sdi.getGridOn to access the current grid configuration. The subsequent sections show the code for each configuration option for time plot grid in the Simulation Data Inspector. To see the result, the example uses a Simulink.sdi.CustomSnapshot object to specify settings for the output of the Simulink.sdi.snapshot function.

snapSettings = Simulink.sdi.CustomSnapshot;
snapSettings.Width = 300;
snapSettings.Height = 300;
Get Initial Grid Setting

Before modifying the grid display preferences, you can save the current configuration to a variable in the workspace in case you want to restore the preferences later.

```matlab
gridInit = Simulink.sdi.getGridOn;
```

**Configure Horizontal Grid Lines**

Show only horizontal grid lines in the Simulation Data Inspector.

```matlab
Simulink.sdi.setGridOn('horizontal')
Simulink.sdi.snapshot('from','custom','settings',snapSettings);
```

**Configure Vertical Grid Lines**

Show only vertical grid lines in the Simulation Data Inspector.
Simulink.sdi.setGridOn('vertical')
Simulink.sdi.snapshot('from','custom','settings',snapSettings);

**Configure No Grid Lines**

Show no grid lines in the Simulation Data Inspector.

Simulink.sdi.setGridOn('off')
Simulink.sdi.snapshot('from','custom','settings',snapSettings);
Configure Horizontal and Vertical Grid Lines

Show horizontal and vertical grid lines in the Simulation Data Inspector.

Simulink.sdi.setGridOn('on')
Simulink.sdi.snapshot('from','custom','settings',snapSettings);
Output Arguments

grid — Time plot grid configuration
'on' (default) | 'horizontal' | 'vertical' | 'off'

Grid configuration for time plots in the Simulation Data Inspector.

- 'on' — Horizontal and vertical grid lines are shown on time plots in the Simulation Data Inspector.
- 'horizontal' — Only horizontal grid lines are shown on time plots in the Simulation Data Inspector.
- 'vertical' — Only vertical grid lines are shown on time plots in the Simulation Data Inspector.
• 'off' — No grid lines are shown on time plots in the Simulation Data Inspector.

Compatibility Considerations

Simulink.sdi.getGridOn return values have changed

Behavior changed in R2019b

Starting in R2019b, the Simulink.sdi.getGridOn return values changed to:

• 'on'
• 'horizontal'
• 'vertical'
• 'off'

In R2019a, the Simulink.sdi.getGridOn returned a logical value:

• true or 1 — Horizontal and vertical grid lines are shown on time plots in the Simulation Data Inspector.
• false or 0 — Horizontal and vertical grid lines are hidden on time plots in the Simulation Data Inspector.

Starting in R2019b, the 'on' return value is equivalent to the true or 1 return, and the 'off' return value is equivalent to the false or 0 return.

See Also

Simulation Data Inspector | Simulink.sdi.clearPreferences |
Simulink.sdi.getBorderOn | Simulink.sdi.getMarkersOn |
Simulink.sdi.getTickLabelsDisplay | Simulink.sdi.getTicksPosition |
Simulink.sdi.setGridOn

Topics

“Create Plots Using the Simulation Data Inspector”
“View Data with the Simulation Data Inspector”

Introduced in R2019a
Simulink.sdi.getMarkersOn

**Package:** Simulink.sdi

Determine if data markers are shown in the Simulation Data Inspector

**Syntax**

```matlab
markersOn = Simulink.sdi.getMarkersOn
```

**Description**

`markersOn = Simulink.sdi.getMarkersOn` returns a logical value indicating whether data markers are displayed on plots in the Simulation Data Inspector.

**Examples**

**Store Marker State**

You can check and store the value of the *Show markers* setting in the Simulation Data Inspector from the command window or in a script.

```matlab
markersOn = Simulink.sdi.getMarkersOn;
```

**Output Arguments**

- `markersOn` — Logical indication of marker state
  - `false` or `0` (default) | `true` or `1`

Logical indication of whether markers are displayed on plots in the Simulation Data Inspector.

- `true` or `1` indicates that markers are displayed.
false or 0 indicates that markers are not displayed.

See Also
Simulation Data Inspector | Simulink.sdi.clearPreferences |
Simulink.sdi.getBorderOn | Simulink.sdi.getGridOn |
Simulink.sdi.getTickLabelsDisplay | Simulink.sdi.getTicksPosition |
Simulink.sdi.setMarkersOn

Topics
“Inspect and Compare Data Programmatically”

Introduced in R2017b
Simulink.sdi.getRun

**Package:** Simulink.sdi

Get a Simulink.sdi.Run object to access data

**Syntax**

```matlab
run = Simulink.sdi.getRun(runID)
```

**Description**

```matlab
run = Simulink.sdi.getRun(runID)
```
returns a Simulink.sdi.Run object that provides access to the data in the run corresponding to the runID. The Simulation Data Inspector assigns run IDs when it creates a run. You can get the run ID for your run using Simulink.sdi.getAllRunIDs or Simulink.sdi.getRunIDByIndex.

**Examples**

**Get a Run Object for Simulation Data**

Many workflows using the Simulation Data Inspector programmatic interface start with acquiring a Simulink.sdi.Run object for your simulation data.

```matlab
% Load and simulate system
load_system('sldemo_fuelsys')
sim('sldemo_fuelsys')

% Get runID for most recent run
runIDs = Simulink.sdi.getAllRunIDs;
runID = runIDs(end);

% Get run object
run = Simulink.sdi.getRun(runID);
```
You can use the Simulink.sdi.Run object to access signal data, add data, and inspect run metadata.

**Plot Signals from a Simulation Run**

This example demonstrates how to access the Simulink.sdi.Run object for a Simulation Data Inspector run created by logging signals. From the Simulink.sdi.Run object you can get Simulink.sdi.Signal objects that you can use to plot the data.

**Create a Simulation Run and Access the Run Object**

The sldemo_fuelsys model includes several signals that are marked for logging. To create a simulation run containing the logged data, simulate the model.

```matlab
sim('ex_vdp');
```

The Simulation Data Inspector keeps track of runs by assigning each run created by simulation, importing data, or opening a session a unique numeric ID. To access the run object for the simulation just performed, use the Simulink.sdi.getAllRunIDs function and take the last run ID in the returned vector.

```matlab
runIDs = Simulink.sdi.getAllRunIDs;
runID = runIDs(end);
```

Once you have the run ID for the run of interest, you can use the Simulink.sdi.getRun function to get the Simulink.sdi.Run object corresponding to the run. You can use the Run object to check metadata associated with the run, including the number of signals in the run.

```matlab
vdpRun = Simulink.sdi.getRun(runID);

vdpRun.SignalCount
ans = int32
    2
```

**Plot Data Using Signal Objects**

Use the getSignalByIndex function to access signals of interest from the Run object, fuelRun.

```matlab
signal1 = getSignalByIndex(vdpRun,1);
signal2 = getSignalByIndex(vdpRun,2);
```

Use the Simulink.sdi.setSubPlotLayout function to specify a 3-by-1 layout.
Simulink.sdi.setSubPlotLayout(2,1)

Before plotting the data, use the `Simulink.sdi.clearAllSubPlots` function to clear any data that is already plotted.

`Simulink.sdi.clearAllSubPlots`

Plot one signal on each subplot. To plot signals on the first subplot, you can set the checked property for the signal. To plot signals on subplots other than the first subplot, use the `plotOnSubPlot` function.

`signal1.Checked = true; plotOnSubPlot(signal2,2,1,true);`

**View the Plotted Data**

To view the plots you just created, open the Simulation Data Inspector using the `Simulink.sdi.view` function.
Input Arguments

runID — Numeric run identifier
scalar

Run ID for the run you want a `Simulink.sdi.Run` object for. The Simulation Data Inspector assigns run IDs when it creates runs. You can get the run ID for a run using `Simulink.sdi.getAllRunIDs` or `Simulink.sdi.getRunIDByIndex`. 
Output Arguments

run — Simulink.sdi.Run object
'Simulink.sdi.Run'

Simulink.sdi.Run object for the run corresponding to the run ID.

See Also
Simulink.sdi.Run | Simulink.sdi.createRun | Simulink.sdi.getAllRunIDs | Simulink.sdi.getRunIDByIndex

Topics
“Inspect and Compare Data Programmatically”

Introduced in R2011b
Simulink.sdi.getRunCount

**Package:** Simulink.sdi

Get number of runs in Simulation Data Inspector repository

**Syntax**

count = Simulink.sdi.getRunCount

**Description**

count = Simulink.sdi.getRunCount returns the number of runs in the Simulation Data Inspector repository. You can use the run count to loop over all runs in the Simulation Data Inspector repository to modify run or signal properties. For example, you could add an absolute tolerance to a signal in every run.

**Examples**

**Modify Parameter for Several Runs**

This example shows how to modify a parameter for all the runs in the Simulation Data Inspector programmatically.

**Generate Runs**

Load the vdp model and mark the x1 and x2 signals for logging. Then, run several simulations.

% Clear all data from the Simulation Data Inspector repository
Simulink.sdi.clear

% Load the model and mark signals of interest for streaming
load_system('vdp')
Simulink.sdi.markSignalForStreaming('vdp/x1',1,'on')
Simulink.sdi.markSignalForStreaming('vdp/x2',1,'on')

% Simulate the model with several Mu values
for gain = 1:5
    gainVal = num2str(gain);
    set_param('vdp/Mu','Gain',gainVal)
    sim('vdp');
end

Use Simulink.sdi.getRunCount to Assign Tolerance to x1 Signals

count = Simulink.sdi.getRunCount;

for a = 1:count
    runID = Simulink.sdi.getRunIDByIndex(a);
    vdpRun = Simulink.sdi.getRun(runID);
    sig = vdpRun.getSignalByIndex(1);
    sig.AbsTol = 0.1;
end

% Open the Simulation Data Inspector to view your data
Simulink.sdi.view

Output Arguments

count — Number of runs
scalar

Number of runs in the Simulation Data Inspector repository.

See Also
Simulink.sdi.Run | Simulink.sdi.Signal | Simulink.sdi.getAllRunIDs | Simulink.sdi.getRun | Simulink.sdi.getRunIDByIndex

Topics
“Inspect and Compare Data Programmatically”

Introduced in R2011b
Simulink.sdi.getRunIDByIndex

**Package:** Simulink.sdi

Use Simulation Data Inspector run index to get run ID

**Syntax**

```matlab
runID = Simulink.sdi.getRunIDByIndex(index)
```

**Description**

`runID = Simulink.sdi.getRunIDByIndex(index)` returns the run ID for the run with the specified index in the Simulation Data Inspector repository.

**Examples**

**Get Run ID for a Simulation**

Many workflows that use the Simulation Data Inspector programmatic interface start with obtaining the ID for a simulation run. This example shows two different methods to use the programmatic interface to get the run ID for a run. You can use the run ID to create a Simulink.sdi.Run object to access run data and metadata, or you can use the run ID for a comparison.

**Simulate a Model to Create a Run**

The model `sldemo_fuelsys` is already configured for logging. When you simulate the model, the Simulation Data Inspector automatically creates a run and assigns it a run ID.

```matlab
% Load and simulate system
load_system('sldemo_fuelsys')
sim('sldemo_fuelsys')
```
Get Run ID Using Simulink.sdi.getAllRunIDs

Simulink.sdi.getAllRunIDs returns an array of all run IDs for runs in the Simulation Data Inspector repository in order, with the most recently created run at the end.

```matlab
% Get runID for most recent run
runIDs = Simulink.sdi.getAllRunIDs;
runID = runIDs(end);
```

Get Run ID Using Simulink.sdi.getRunIDByIndex

You can also use Simulink.sdi.getRunCount and Simulink.sdi.getRunIDByIndex to get the run ID for a run. This method is useful if you also want to use count as a counting variable to index through the runs in the Simulation Data Inspector repository.

```matlab
count = Simulink.sdi.getRunCount;
runID = Simulink.sdi.getRunIDByIndex(count);
```

Input Arguments

- **index** — Run index in Simulation Data Inspector
  integer
  Positive, whole number index of the run in the Simulation Data Inspector repository.
  Example: 3

Output Arguments

- **runID** — Numeric run identifier
  scalar
  Numeric run identification assigned by the Simulation Data Inspector.

See Also

Simulink.sdi.Run | Simulink.sdi.compareRuns | Simulink.sdi.copyRun | Simulink.sdi.deleteRun | Simulink.sdi.getRun | Simulink.sdi.getRunCount | Simulink.sdi.isValidRunID
Topics
"Inspect and Compare Data Programmatically"

Introduced in R2011b
Simulink.sdi.getRunNamingRule

**Package:** Simulink.sdi

Get the Simulation Data Inspector rule for naming runs

**Syntax**

namingRule = Simulink.sdi.getRunNamingRule

**Description**

namingRule = Simulink.sdi.getRunNamingRule returns the run naming rule as a character vector. The run naming rule can contain one or more tokens that update for each run, for example, `<run_index>`. The run naming rule applies to runs automatically created through simulating a model in Simulink.

**Examples**

**Modify Run Naming Rule Then Restore Default**

This example shows how to use the Simulation Data Inspector API to modify the Simulation Data Inspector run naming rule, check a run's name, restore default preferences, and check the run naming rule.

```matlab
% Load model
load_system('sldemo_fuelsys')

% Modify run naming rule
Simulink.sdi.setRunNamingRule('<model_name> Run <run_index>')

% Simulate system
sim('sldemo_fuelsys')

% Check run name
```
runIDs = Simulink.sdi.getAllRunIDs;
runID = runIDs(end);
fuelRun = Simulink.sdi.getRun(runID);
fuelRun.name

ans =
'sldemo_fuelsys Run 1'

% Clear preferences to reset the run naming rule
Simulink.sdi.clearPreferences

% Check run naming rule
Simulink.sdi.getRunNamingRule

ans =
'Run <run_index>: <model_name>'

Output Arguments

namingRule — Naming rule for Simulation Data Inspector runs
character vector

Character vector that specifies the naming rule the Simulation Data Inspector uses to name the runs automatically created through simulating a Simulink model. The run naming rule can contain any of the following tokens that represent information pulled for each run:

- <run_index> – Run's index in the Simulation Data Inspector repository.
- <model_name> - Name of the model simulated to create the run.
- <time_stamp> - Start time for the simulation that created the run.
- <sim_mode> – Simulation mode used for the simulation that created the run.

Alternatives

You can view the run naming rule using the Simulation Data Inspector UI. You can find the New Run options under the Simulation Data Inspector Preferences menu.
See Also
Simulink.sdi.Run | Simulink.sdi.clearPreferences | Simulink.sdi.resetRunNamingRule | Simulink.sdi.setRunNamingRule

Topics
“Inspect and Compare Data Programmatically”

Introduced in R2015a
**Simulink.sdi.getSignal**

**Package:** Simulink.sdi

Get Simulink.sdi.Signal object for a signal

**Syntax**

\[
\text{signalObj} = \text{Simulink.sdi.getSignal}(\text{sigID})
\]

**Description**

\[
\text{signalObj} = \text{Simulink.sdi.getSignal}(\text{sigID}) \text{ returns a Simulink.sdi.Signal object for the signal in the Simulation Data Inspector repository that corresponds to the signal ID, sigID. The Simulink.sdi.Signal object manages signal data and metadata and allows you to view and modify signal properties.}
\]

**Examples**

**Create a Simulation Data Inspector Run and Access Signal Data**

This example shows how to access signal data when you create a run in the Simulation Data Inspector.

**Generate Data for Run**

For this example, create timeseries data for sine and cosine signals.

\[
% \text{Create timeseries workspace data}
\text{time} = \text{linspace}(0, 20, 101);
\]

\[
\text{sine_vals} = \sin(2\pi/5*\text{time});
\]

\[
\text{sine_ts} = \text{timeseries}('sine_vals',\text{time});
\]

\[
\text{sine_ts.Name} = 'Sine, T=5';
\]
\[
\text{cos_vals} = \cos(2\pi/8\times\text{time});
\]
\[
\text{cos_ts} = \text{timeseries}([\text{cos_vals, time}]);
\]
\[
\text{cos_ts.Name} = '\text{Cosine, T=8}';
\]

**Create a Run and Return Signal IDs**

You can use the `Simulink.sdi.createRun` syntax with multiple return arguments to get the signal IDs more directly instead of accessing the signal IDs through a `Simulink.sdi.Run` object.

\[
[\text{runID, runIndex, sigIDs}] = \text{Simulink.sdi.createRun('Sinusoids','vars',...}
\text{sine_ts,cos_ts});
\]

\[
\text{cosID} = \text{sigIDs}(2);
\]
\[
\text{cosSig} = \text{Simulink.sdi.getSignal(cosID)};
\]

**Modify Signal Properties and View in the Simulation Data Inspector**

You can use the `Simulink.sdi.Signal` object to view and modify signal properties and to plot signals in the Simulation Data Inspector.

\[
\text{cosSig.Checked} = \text{true};
\]
\[
\text{cosSig.AbsTol} = 0.05;
\]
\[
\text{Simulink.sdi.view}
\]
\[
\text{cosSig.Name}
\]

**Input Arguments**

- **sigID** — Signal ID
  scalar

  Signal identifier. The Simulation Data Inspector assigns signal IDs to signals when a run is created. You can get the signal ID for a signal as a return from `Simulink.sdi.createRun` or using the `Simulink.sdi.Run` object's methods.

**Output Arguments**

- **signalObj** — `Simulink.sdi.Signal` object
  
  'Simulink.sdi.Signal' object

  Simulink.sdi.Signal object for the signal corresponding to `sigID`. 
See Also
Simulink.sdi.Run | Simulink.sdi.Signal | Simulink.sdi.createRun |
getSignalIDByIndex

Topics
“Inspect and Compare Data Programatically”

Introduced in R2011b
Simulink.sdi.getTickLabelsDisplay

Get tick mark label setting for time plots

Syntax

tickLabels = Simulink.sdi.getTickLabelsDisplay

Description

tickLabels = Simulink.sdi.getTickLabelsDisplay returns the current setting for displayed tick mark labels on time plots in the Simulation Data Inspector. The return value corresponds to the setting for Tick labels in the Simulation Data Inspector Time Plot Settings. The setting applies to all time plots in the layout. By default, the Simulation Data Inspector displays tick mark labels for both axes on time plots.

Examples

Configure Tick Mark Label Visibility for Time Plots

You can use the Simulink.sdi.setTickLabelsDisplay function to configure the visibility of tick mark labels for the t- and y-axes on time plots in the Simulation Data Inspector. By default, the Simulation Data Inspector displays tick marks for both axes.

This example starts by showing how to use the Simulink.sdi.getTickLabelsDisplay function to access the current tick mark label visibility. Subsequent sections show the code to specify each available option for tick mark label visibility on time plots in the Simulation Data Inspector. To see the result of each configuration, the example generates an image using the Simulink.sdi.snapshot function with settings specified by a Simulink.sdi.CustomSnapshot object.

snapSettings = Simulink.sdi.CustomSnapshot;
snapSettings.Width = 300;
snapSettings.Height = 300;
Get Current Tick Mark Label Visibility

Before modifying the visibility of the tick mark labels on time plots, you can save the current configuration to a variable in the workspace in case you want to restore the preference later.

```matlab
initTickLabels = Simulink.sdi.getTickLabelsDisplay;
```

Show Only t-Axis Tick Mark Labels

Show the tick mark labels for the t-Axis.

```matlab
Simulink.sdi.setTickLabelsDisplay('t-axis');
Simulink.sdi.snapshot('from','custom','settings',snapSettings);
```
Show Only y-Axis Tick Mark Labels

Show the tick mark labels for the y-axis.

```matlab
Simulink.sdi.setTickLabelsDisplay('y-axis')
Simulink.sdi.snapshot('from','custom','settings',snapSettings);
```

Hide Tick Mark Labels

You can hide the tick mark labels for both axes.

```matlab
Simulink.sdi.setTickLabelsDisplay('none')
Simulink.sdi.snapshot('from','custom','settings',snapSettings);
```
Show All Tick Mark Labels

By default, the Simulation Data Inspector shows tick mark labels for both axes on time plots.

```matlab
Simulink.sdi.setTickLabelsDisplay('all')
Simulink.sdi.snapshot('from','custom','settings',snapSettings);
```
Output Arguments

tickLabels — Time plot axes that display tick mark labels
'all' | 't-axis' | 'y-axis' | 'none'

Current setting for tick mark labels displayed on time plots in the Simulation Data Inspector:

- 'all' — Tick mark labels displayed for both time plot axes.
- 't-axis' — Tick mark labels displayed only on the t-axis for time plots.
- 'y-axis' — Tick mark labels displayed only on the y-axis for time plots.
- 'none' — No tick mark labels displayed on time plots.
See Also
Simulink.sdi.clearPreferences | Simulink.sdi.getBorderOn |
Simulink.sdi.getGridOn | Simulink.sdi.getMarkersOn |
Simulink.sdi.getTicksPosition | Simulink.sdi.setTickLabelsDisplay

Topics
“Create Plots Using the Simulation Data Inspector”

Introduced in R2019b
Simulink.sdi.getTicksPosition

Get tick mark position setting for time plots

Syntax

ticksPos = Simulink.sdi.getTicksPosition

Description

ticksPos = Simulink.sdi.getTicksPosition returns the current setting for the position of tick marks on time plots in the Simulation Data Inspector. The return value corresponds to the setting for the Ticks setting in the Simulation Data Inspector Time Plot Settings. The setting applies to all time plots in the layout. By default, the Simulation Data Inspector displays tick marks on the outside of the plot area for time plots.

Examples

Configure Position for Tick Marks on Time Plots

You can use Simulink.sdi.setTicksPosition function to specify the position for tick marks on time plots in the Simulation Data Inspector. By default, the Simulation Data Inspector displays tick marks outside of the plot area for time plots.

This example starts by showing how to use the Simulink.sdi.getTicksPosition function to access the current position of tick marks. Subsequent sections show the code to specify each configuration option for the position of tick marks on time plots in the Simulation Data Inspector. To see the result, the example generates an image using Simulink.sdi.snapshot with settings specified by a Simulink.sdi.CustomSnapshot object.

snapSettings = Simulink.sdi.CustomSnapshot;
snapSettings.Width = 300;
snapSettings.Height = 300;
Get Current Tick Mark Position

Before modifying the position of tick marks on time plots, you can save the current configuration to a variable in the workspace in case you want to restore the preference later.

initTickPos = Simulink.sdi.getTicksPosition;

Position Tick Marks Inside Plot Area

Position the tick marks inside the plot area to increase the amount of space used by the plot area.

Simulink.sdi.setTicksPosition('inside')
Simulink.sdi.snapshot('from','custom','settings',snapSettings);
**Hide Tick Marks on Time Plots**

You can hide the tick marks for time plots in the Simulation Data Inspector. Hiding the tick marks expands the plot area.

```matlab
Simulink.sdi.setTicksPosition('none')
Simulink.sdi.snapshot('from','custom','settings',snapSettings);
```

**Position Tick Marks Outside Plot Area**

By default, the Simulation Data Inspector displays tick marks outside of the plot area for time plots.

```matlab
Simulink.sdi.setTicksPosition('outside')
Simulink.sdi.snapshot('from','custom','settings',snapSettings);
```
Output Arguments

ticksPos — Current setting for tick mark position on time plots
'outside' | 'inside' | 'none'

Current setting for the position of tick marks on time plots in the Simulation Data Inspector.

See Also
Simulink.sdi.clearPreferences | Simulink.sdi.getBorderOn |
Simulink.sdi.getGridOn | Simulink.sdi.getMarkersOn |
Simulink.sdi.getTickLabelsDisplay | Simulink.sdi.setTicksPosition
Topics
“Create Plots Using the Simulation Data Inspector”

Introduced in R2019b
Simulink.sdi.isPCTSupportEnabled

Determine status and mode for Parallel Computing Toolbox support

Syntax

[enabled, mode] = Simulink.sdi.isPCTSupportEnabled

Description

[enabled, mode] = Simulink.sdi.isPCTSupportEnabled returns enabled, a logical indication of whether support for the Parallel Computing Toolbox is enabled, and mode, the mode of support enabled.

Examples

Check Status of Parallel Worker Support

Before running code that depends on whether automatic import of runs created by parallel workers is enabled, you can use the Simulink.sdi.isPCTSupportEnabled function to check the support status. The default behavior for the Simulation Data Inspector enables parallel worker support in local mode. In local mode, only runs created on local workers automatically import into the Simulation Data Inspector.

[enabled, mode] = Simulink.sdi.isPCTSupportEnabled

enabled =

    logical

    1

mode =
'local'

Output Arguments

enabled — Logical indicator of parallel worker support
logical

Logical indication of parallel worker support.

• 1 indicates that support for parallel workers is enabled.
• 0 indicates that support for parallel workers is not enabled.

mode — Parallel worker support mode
local (default)

Mode of Parallel Computing Toolbox support.

• 'local' — Runs generated on local workers automatically import to the Simulation Data Inspector.
• 'none' — Parallel worker support is disabled.
• 'all' — Runs created from local and remote workers automatically import to the Simulation Data Inspector.
• 'manual' — Support for manual import of runs created by parallel workers using the Simulink.sdi.sendWorkerRunToClient function.

See Also
Simulink.sdi.WorkerRun | Simulink.sdi.enablePCTSupport | Simulink.sdi.sendWorkerRunToClient

Topics
"Inspect and Compare Data Programmatically”

Introduced in R2017b
Simulink.sdi.isValidRunID

**Package:** Simulink.sdi

Determine whether a run ID is valid

**Syntax**

```matlab
valid = Simulink.sdi.isValidRunID(runID)
```

**Description**

```matlab
valid = Simulink.sdi.isValidRunID(runID)
```

returns **true** if **runID** corresponds to a run in the Simulation Data Inspector repository.

**Examples**

**Check Run ID Validity**

This example shows how to check whether a run ID is valid. You can use `Simulink.sdi.isValidRunID` to ensure you have valid data throughout your script.

**Create a Simulation Run**

Simulate the model `sldemo_fuelsys` to create a run in the Simulation Data Inspector, and use `Simulink.sdi.getAllRunIDs` to get its run ID.

```matlab
% Simulate model
load_system('sldemo_fuelsys')
sim('sldemo_fuelsys')

% Get run ID
runIDs = Simulink.sdi.getAllRunIDs;
runID = runIDs(end);
```
Check Run ID Validity

Check to verify that the Simulation Data Inspector has a run corresponding to the run ID.

```matlab
Simulink.sdi.isValidRunID(runID)
```

ans = logical

1

Delete the Run and Check Validity

You can delete runs to clear out memory space or clean up the Simulation Data Inspector UI. When you delete a run, the run ID for that run becomes invalid.

```matlab
Simulink.sdi.deleteRun(runID)
Simulink.sdi.isValidRunID(runID)
```

ans = logical

0

Input Arguments

runID — Simulation Data Inspector run identifier
scalar

Unique numeric identification for the run. The Simulation Data Inspector assigns run IDs when it creates runs. You can get the run ID for your run using Simulink.sdi.getAllRunIDs or Simulink.sdi.getRunIDByIndex.

Output Arguments

valid — Run validity indicator
logical

Run validity indicator. When valid is true, the runID is valid. When valid is false, the runID is invalid.
See Also
Simulink.sdi.compareRuns | Simulink.sdi.createRun |
Simulink.sdi.deleteRun | Simulink.sdi.getAllRunIDs |
Simulink.sdi.getRunIDByIndex

Topics
“Inspect and Compare Data Programmatically”

Introduced in R2011b
Simulink.sdi.load

**Package:** Simulink.sdi

Load a Simulation Data Inspector session or view

**Syntax**

valid = Simulink.sdi.load(fileName)

**Description**

valid = Simulink.sdi.load(fileName) loads the data in the MLDATX-file or MAT-file specified by fileName and returns 1 when fileName is a valid file. A return value of 0 indicates that the file specified by fileName is invalid and cannot be loaded into the Simulation Data Inspector. You can use Simulink.sdi.load to load Simulation Data Inspector sessions and views. A view includes preferences and visualization options but does not save data. A session saves data along with preference selections and plot configurations.

**Examples**

**Save a Simulation Data Inspector Session**

This example shows how to create, save, and load a Simulation Data Inspector session. The example uses data logging to populate the Simulation Data Inspector with data and then uses the Simulation Data Inspector's programmatic interface to create plots to visualize the data. After saving the data and visualization settings in a session, the Simulation Data Inspector repository is emptied in order to demonstrate how to load the session.
Create Simulation Data

This example logs the Stick, alpha, \( \text{rad} \), and \( q \), rad/sec signals to generate simulation data using the model `slexAircraftExample` and creates two runs. The first uses a sine input, and the second has a square wave input.

```matlab
% Ensure you start with an empty Simulation Data Inspector repository
Simulink.sdi.clear

% Load system
load_system('slexAircraftExample')

% Configure signals to log
Simulink.sdi.markSignalForStreaming('slexAircraftExample/Pilot',1,'on')
Simulink.sdi.markSignalForStreaming('slexAircraftExample/Aircraft Dynamics Model',3,'on')
Simulink.sdi.markSignalForStreaming('slexAircraftExample/Aircraft Dynamics Model',4,'on')

% Change Pilot signal to sine
set_param('slexAircraftExample/Pilot','WaveForm','sine')

% Simulate model
sim('slexAircraftExample')

% Change Pilot signal to square
set_param('slexAircraftExample/Pilot','WaveForm','square')

% Simulate model
sim('slexAircraftExample')

Access Simulation Data

Use the Simulation Data Inspector programmatic interface to access the simulation data so you can create plots to visualize the signals.

```matlab
% Get run objects
runIDs = Simulink.sdi.getAllRunIDs;
sineRunID = runIDs(end-1);
squareRunID = runIDs(end);
sineRun = Simulink.sdi.getRun(sineRunID);
squareRun = Simulink.sdi.getRun(squareRunID);

% Get signal objects
sineOut = sineRun.getSignalByIndex(1);
sineIn = sineRun.getSignalByIndex(3);
```
squareOut = squareRun.getSignalByIndex(1);
squareIn = squareRun.getSignalByIndex(3);

**Create Plots in the Simulation Data Inspector**

Use the programmatic interface to visualize the signal data from the two simulation runs. You can set the plot layout and plot signals on specific subplots.

% Set subplot layout
Simulink.sdi.setSubPlotLayout(2,1)

% Plot sine data on top plot
sineIn.plotOnSubPlot(1,1,true)
sineOut.plotOnSubPlot(1,1,true)

% Plot square wave data on bottom plot
squareIn.plotOnSubPlot(2,1,true)
squareOut.plotOnSubPlot(2,1,true)

**Save a Simulation Data Inspector Session**

First, view the plots you just created. Then, save the Simulation Data Inspector session as an MLDATX-file to recover your data along with your preference selections and plots.

% View the visualized data in the Simulation Data Inspector
Simulink.sdi.view

% Save the Simulation Data Inspector session
Simulink.sdi.save('myData.mldatx')

**Load a Simulation Data Inspector Session**

First, clear the Simulation Data Inspector repository with Simulink.sdi.clear and reset visualization settings with Simulink.sdi.clearPreferences. Then, you can load the session to see how the data and settings are preserved.

% Clear Simulation Data Inspector repository and preferences
Simulink.sdi.clear
Simulink.sdi.clearPreferences
% Load session file to view data
Simulink.sdi.load('myData.mldatx');

**Input Arguments**

`fileName` — Name of file to load
character vector

Name of the file to load with the session or view data.
Example: 'myData.mldatx'
Example: 'myData.mat'

**Output Arguments**

`valid` — File validity indicator
logical

Validity indicator for the file. When the file specified by `fileName` is valid, `valid` is 1. A valid value of 0 indicates an invalid file.

**See Also**

`Simulink.sdi.close` | `Simulink.sdi.createRun` | `Simulink.sdi.save`

**Topics**

“Inspect and Compare Data Programmatically”
“View Data with the Simulation Data Inspector”

**Introduced in R2011b**
Simulink.sdi.markSignalForStreaming

Package: Simulink.sdi

Turn logging on or off for a signal

Syntax

Simulink.sdi.markSignalForStreaming(block,portIndex,log)
Simulink.sdi.markSignalForStreaming(portHandle,log)
Simulink.sdi.markSignalForStreaming(lineHandle,log)

Description

Simulink.sdi.markSignalForStreaming(block,portIndex,log) marks the signal on the specified portIndex of the specified block for logging when you specify log as 'on'. To stop logging a signal, specify log as 'off'.

Simulink.sdi.markSignalForStreaming(portHandle,log) marks the signal on the port specified by portHandle for logging when you specify log as 'on'. To stop logging a signal, specify log as 'off'.

Simulink.sdi.markSignalForStreaming(lineHandle,log) marks the signal with the specified lineHandle for logging when you specify log as 'on'. To stop logging a signal, specify log as 'off'.

Examples

Compare Signals Within a Simulation Run

This example uses the slexAircraftExample model to demonstrate the comparison of the input and output signals for a control system. The example marks the signals for streaming then gets the run object for a simulation run. Signal IDs from the run object specify the signals to be compared.
% Load model slexAircraftExample and mark signals for streaming
load_system('slexAircraftExample')
Simulink.sdi.markSignalForStreaming('slexAircraftExample/Pilot',1,'on')
Simulink.sdi.markSignalForStreaming('slexAircraftExample/Aircraft Dynamics Model',4,'on')

% Simulate model slexAircraftExample
out = sim('slexAircraftExample');

% Get run IDs for most recent run
allIDs = Simulink.sdi.getAllRunIDs;
runID = allIDs(end);

% Get Run object
aircraftRun = Simulink.sdi.getRun(runID);

% Get signal IDs
signalID1 = getSignalIDByIndex(aircraftRun,1);
signalID2 = getSignalIDByIndex(aircraftRun,2);

if (isValidSignalID(aircraftRun,signalID1))
    % Change signal tolerance
    signal1 = Simulink.sdi.getSignal(signalID1);
    signal1.AbsTol = 0.1;
end

if (isValidSignalID(aircraftRun,signalID1) && isValidSignalID(aircraftRun,signalID2))
    % Compare signals
    sigDiff = Simulink.sdi.compareSignals(signalID1,signalID2);

    % Check whether signal result status
    match = sigDiff.Status
end

match =
OutOfRange

**Mark Signals for Logging with Port Handles**

This example shows how to mark signals for logging using port handles.
Load Model and Mark Signals for Streaming

User `get_param` to get the port handles for the blocks with your signals of interest. Then, use the handle to mark the desired signals for logging.

```matlab
load_system('vdp')

% Get port handles
x1_handles = get_param('vdp/x1','PortHandles');
x1 = x1_handles.Outport(1);
x2_handles = get_param('vdp/x2','PortHandles');
x2 = x2_handles.Outport(1);

% Mark signals for streaming
Simulink.sdi.markSignalForStreaming(x1,'on');
Simulink.sdi.markSignalForStreaming(x2,'on');
```

Simulate Model and View Signals in the Simulation Data Inspector

Simulate the model and then open the Simulation Data Inspector to view the logged signals.

```matlab
sim('vdp');
Simulink.sdi.view
```

Mark Signals for Logging with Line Handles

This example shows how to mark signals for logging using their line handles.

Load System and Mark Signals for Logging

Load a model and use `get_param` to get handles for the signals in the model. Then, use the line handles to mark signals of interest for logging.

```matlab
load_system('slexAircraftExample')
lines = get_param('slexAircraftExample','Lines');
sig1handle = lines(1).Handle;
sig2handle = lines(2).Handle;
```
Simulink.sdi.markSignalForStreaming(sig1handle,'on')
Simulink.sdi.markSignalForStreaming(sig2handle,'on')

**Simulate Model and View Signals**

Simulate the model and view the signals marked for logging in the Simulation Data Inspector.

```matlab
sim('slexAircraftExample')
```

Simulink.sdi.view

**Input Arguments**

**block** — Source block path or handle
character vector

Block path for the block with the desired signal connected to one of its outputs.

Example: 'slexAircraftExample/Pilot'

**portIndex** — Source block output port index
integer

Index of the port connected to the signal you want to mark for streaming.

Example: 1

**log** — Logging state
'on' | 'off'

Logging state desired for signal.

- 'on' -- Turn logging on for a signal.
- 'off' -- Turn logging off for a signal.

**portHandle** — Source block output port handle
handle

Port handle for the source block's output port that connects to the signal.

Example: x1_handles.Outport(1)
**lineHandle — Signal line handle**

handle

Line handle for the signal.

Example: `lines(1).Handle`

**See Also**

`Simulink.HMI.InstrumentedSignals` |
`Simulink.sdi.createRunOrAddToStreamedRun` |
`Simulink.sdi.getAllRunIDs` |
`Simulink.sdi.getRunIDByIndex`

**Topics**

“Inspect and Compare Data Programmatically”

“View Data with the Simulation Data Inspector”

**Introduced in R2015b**
Simulink.sdi.report

Package: Simulink.sdi

Generate a Simulation Data Inspector report

Syntax

Simulink.sdi.report
Simulink.sdi.report(Name,Value)

Description

Simulink.sdi.report creates a Simulation Data Inspector report of the plotted data in the Inspect pane of the Simulation Data Inspector.

Simulink.sdi.report(Name,Value) uses additional options specified by one or more Name,Value pair arguments to generate a report of the specified view in the Simulation Data Inspector.

Examples

Generate a Simulation Data Inspector Report Programmatically

This example shows how to create reports using the Simulation Data Inspector programmatic interface. You can create a report for plotted signals in the Inspect pane or for comparison data in the Compare pane. This example first generates data by simulating a model, then shows how to create an Inspect Signals report. To run the example exactly as shown, ensure that the Simulation Data Inspector repository starts empty with the Simulink.sdi.clear function.

Generate Data

This example generates data using model ex_sldemo_absbrake with two different desired slip ratios.
% Ensure Simulation Data Inspector is empty
Simulink.sdi.clear

% Open model
load_system('ex_sldemo_absbrake')

% Set slip ratio and simulate model
set_param('ex_sldemo_absbrake/Desired relative slip','Value','0.24')
sim('ex_sldemo_absbrake');

% Set new slip ratio and simulate model again
set_param('ex_sldemo_absbrake/Desired relative slip','Value','0.25')
sim('ex_sldemo_absbrake');

---

**Plot Signals in the Inspect Pane**

The Inspect Signals report includes all signals plotted in the graphical viewing area of the Inspect pane and all displayed metadata for the plotted signals.

% Get Simulink.sdi.Run objects
runIDs = Simulink.sdi.getAllRunIDs;
runID1 = runIDs(end-1);
runID2 = runIDs(end);

run1 = Simulink.sdi.getRun(runID1);
run2 = Simulink.sdi.getRun(runID2);

% Get Simulink.sdi.Signal objects for slp signal
run1_slp = run1.getSignalByIndex(4);
run2_slp = run2.getSignalByIndex(4);

% Plot slp signals
run1_slp.plotOnSubPlot(1, 1, true)
run2_slp.plotOnSubPlot(1, 1, true)

---

**Create a Report of Signals Plotted in Inspect Pane**

You can include more data in the report by adding more columns using the Simulation Data Inspector UI, or you can specify the information you want in the report programmatically with Name-Value pairs and the enumeration class Simulink.sdi.SignalMetaData. This example shows how to specify the data in the report programmatically.

% Specify report parameters
reportType = 'Inspect Signals';
reportName = 'Data_Report.html';

signalMetadata = [Simulink.sdi.SignalMetaData.Run, ...
                Simulink.sdi.SignalMetaData.Line, ...
                Simulink.sdi.SignalMetaData.BlockName, ...
                Simulink.sdi.SignalMetaData.SignalName];

Simulink.sdi.report('ReportToCreate', reportType, 'ReportOutputFile', ...
                     reportName, 'ColumnsToReport', signalMetadata);

### Input Arguments

#### Name-Value Pair Arguments

Specify optional comma-separated pairs of Name, Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside quotes. You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

Example: 'ReportToCreate','Compare Runs','ShortenBlockPath',true

**ReportToCreate — Information to include in the report**

Inspect Signals' (default) | 'Compare Runs'

Simulation Data Inspector pane to capture in the report.

- 'Inspect Signals' -- Create a report of the information on the Inspect pane.
- 'Compare Runs' -- Create a report of the information on the Compare pane.

Example: 'ReportToCreate','Inspect Signals'
Example: 'ReportToCreate','Compare Runs'

**ReportOutputFolder — Folder where report is saved**

<current working folder>/slprj/sdi (default) | path

Folder where the report is saved. Specify the path to the folder where you want to save the report as a character vector.

Example: 'ReportOutputFolder','C:\Users\user1\Desktop'
ReportOutputFile — Report file name
'SDI_report.html' (default) | character vector

File name for report.
Example: 'ReportOutputFile','MyReport.html'

PreventOverwritingFile — Report overwrite prevention
true (default) | false

File overwrite protection for the report. File overwrite protection prevents the Simulation Data Inspector from overwriting an existing file by appending the file name with a number that increments for subsequent reports. When you disable file overwrite protection, the Simulation Data Inspector overwrites the existing report file unless you specify a unique file name.

- true enables file overwrite protection.
- false disables file overwrite protection.

Example: 'PreventOverwritingFile',true
Example: 'PreventOverwritingFile',false

ColumnsToReport — Signal metadata to include in report
array

Signal metadata to include in report. By default, the Inspect Signals report includes the block path, name, line style and color, and data source parameters for each plotted signal. The Compare Runs report includes the signal name, absolute tolerance, relative tolerance, and maximum difference metadata by default.

Specify metadata to include as an array, using the enumeration class Simulink.sdi.SignalMetaData. For example, to include the name of the simulation run and signal name, create an array like signal_metadata:

```matlab
signal_metadata = [Simulink.sdi.SignalMetaData.Run,...
                   Simulink.sdi.SignalMetaData.SignalName];
```

Then, specify ColumnsToReport as signal_metadata in the name-value pair:

```matlab
Simulink.sdi.report('ColumnsToReport',signal_metadata)
```

The table summarizes the metadata available for Inspect Signals report.
<table>
<thead>
<tr>
<th><strong>Column Value</strong></th>
<th><strong>Description</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>SignalName (default)</td>
<td>Signal name</td>
</tr>
<tr>
<td>Line (default)</td>
<td>Signal line style and color</td>
</tr>
<tr>
<td>SID</td>
<td>Simulink identifier</td>
</tr>
<tr>
<td></td>
<td>For more information about SIDs, see “Locate Diagram Components Using Simulink Identifiers”</td>
</tr>
<tr>
<td>Units</td>
<td>Signal measurement units</td>
</tr>
<tr>
<td>SigDataType</td>
<td>Signal data type</td>
</tr>
<tr>
<td>SigSampleTime</td>
<td>Method used to sample the signal</td>
</tr>
<tr>
<td>Model</td>
<td>Name of the model that generated the signal</td>
</tr>
<tr>
<td>BlockName</td>
<td>Name of the source block for the signal</td>
</tr>
<tr>
<td>BlockPath</td>
<td>Path to the source block for the signal</td>
</tr>
<tr>
<td>Port</td>
<td>Index of the signal on the output port of its block</td>
</tr>
<tr>
<td>Dimensions</td>
<td>Dimensions of the matrix containing the signal</td>
</tr>
<tr>
<td>Channel</td>
<td>Index of signal within matrix</td>
</tr>
<tr>
<td>Run</td>
<td>Name of the simulation run containing the signal</td>
</tr>
<tr>
<td>AbsTol</td>
<td>Absolute tolerance for the signal</td>
</tr>
<tr>
<td>RelTol</td>
<td>Relative tolerance for the signal</td>
</tr>
<tr>
<td>OverrideGlobalTol</td>
<td>Property that specifies whether signal tolerances take priority over global tolerances</td>
</tr>
<tr>
<td>TimeTol</td>
<td>Time tolerance for the signal</td>
</tr>
<tr>
<td>InterpMethod</td>
<td>Interpolation method</td>
</tr>
<tr>
<td>SyncMethod</td>
<td>Synchronization method used to coordinate signals for comparison</td>
</tr>
<tr>
<td>Column Value</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>TimeSeriesRoot</td>
<td>Name of the variable associated with the signal for signals imported from the MATLAB workspace</td>
</tr>
<tr>
<td>TimeSource</td>
<td>Name of the array containing the time data for signals imported from the MATLAB workspace</td>
</tr>
<tr>
<td>DataSource</td>
<td>Name of the array containing the signal data for signals imported from the MATLAB workspace</td>
</tr>
</tbody>
</table>

The table provides a summary of the metadata available for the Compare Runs report.

<table>
<thead>
<tr>
<th>Column Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Result (default)</td>
<td>Pass/fail result of the signal comparison between the Baseline and Compare To runs</td>
</tr>
<tr>
<td>Line1</td>
<td>Line style and color for the Baseline signal</td>
</tr>
<tr>
<td>Line2</td>
<td>Line style and color for the Compare To signal</td>
</tr>
<tr>
<td>AbsTol (default)</td>
<td>Absolute tolerance for the Baseline signal</td>
</tr>
<tr>
<td>RelTol (default)</td>
<td>Relative tolerance for the Baseline signal</td>
</tr>
<tr>
<td>MaxDifference</td>
<td>The maximum difference between the Baseline and Compare To signals</td>
</tr>
<tr>
<td>OverrideGlobalTol</td>
<td>Property that specifies whether the Baseline signal tolerances take priority over global tolerances</td>
</tr>
<tr>
<td>TimeTol</td>
<td>Time tolerance for the Baseline signal</td>
</tr>
<tr>
<td>SignalName1</td>
<td>Signal name from the Baseline run</td>
</tr>
<tr>
<td>SignalName2</td>
<td>Signal name from the Compare To run</td>
</tr>
<tr>
<td>Units1</td>
<td>Measurement units for the signal in the Baseline run</td>
</tr>
<tr>
<td>Units2</td>
<td>Measurement units for the signal in the Compare To run</td>
</tr>
<tr>
<td>Column Value</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>SigDataType1</td>
<td>The data type for the signal in the <strong>Baseline</strong> run</td>
</tr>
<tr>
<td>SigDataType2</td>
<td>The data type for the signal in the <strong>Compare To</strong> run</td>
</tr>
<tr>
<td>SigSampleTime1</td>
<td>Method used to sample the signal in the <strong>Baseline</strong> run</td>
</tr>
<tr>
<td>SigSampleTime2</td>
<td>Method used to sample the signal in the <strong>Compare To</strong> run</td>
</tr>
<tr>
<td>Run1</td>
<td>Name of the <strong>Baseline</strong> run</td>
</tr>
<tr>
<td>Run2</td>
<td>Name of the <strong>Compare To</strong> run</td>
</tr>
<tr>
<td>AlignedBy (default)</td>
<td>Signal alignment method used to correlate <strong>Baseline</strong> and <strong>Compare To</strong> signals</td>
</tr>
<tr>
<td>Model1</td>
<td>Name of the model that generated the <strong>Baseline</strong> signals</td>
</tr>
<tr>
<td>Model2</td>
<td>Name of the model that generated the <strong>Compare To</strong> signals</td>
</tr>
<tr>
<td>BlockName1</td>
<td>Name of the source block for the <strong>Baseline</strong> signal</td>
</tr>
<tr>
<td>BlockName2</td>
<td>Name of the source block for the <strong>Compare To</strong> signal</td>
</tr>
<tr>
<td>BlockPath1</td>
<td>Path to the source block for the <strong>Baseline</strong> signal</td>
</tr>
<tr>
<td>BlockPath2</td>
<td>Path to the source block for the <strong>Compare To</strong> signal</td>
</tr>
<tr>
<td>Port1</td>
<td>Index of the <strong>Baseline</strong> signal on the output port of its block</td>
</tr>
<tr>
<td>Port2</td>
<td>Index of the <strong>Compare To</strong> signal on the output port of its block</td>
</tr>
<tr>
<td>Dimensions1</td>
<td>Dimensions of the matrix containing the <strong>Baseline</strong> signal</td>
</tr>
<tr>
<td>Dimensions2</td>
<td>Dimensions of the matrix containing the <strong>Compare To</strong> signal</td>
</tr>
<tr>
<td>Column Value</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Channel1</td>
<td>Index of the Baseline within its matrix</td>
</tr>
<tr>
<td>Channel2</td>
<td>Index of the Compare To within its matrix</td>
</tr>
<tr>
<td>InterpMethod</td>
<td>Interpolation method for the Baseline signal</td>
</tr>
<tr>
<td>SyncMethod</td>
<td>Synchronization method for the Baseline signal</td>
</tr>
<tr>
<td>TimeSeriesRoot1</td>
<td>Name of the variable associated with the Baseline signal for signals imported from the MATLAB workspace</td>
</tr>
<tr>
<td>TimeSeriesRoot2</td>
<td>Name of the variable associated with the Compare To signal for signals imported from the MATLAB workspace</td>
</tr>
<tr>
<td>TimeSource1</td>
<td>Name of the array containing the Baseline time data for signals imported from the MATLAB workspace</td>
</tr>
<tr>
<td>TimeSource2</td>
<td>Name of the array containing the Compare To time data for signals imported from the MATLAB workspace</td>
</tr>
<tr>
<td>DataSource1</td>
<td>Name of the array containing the Baseline signal data for signals imported from the MATLAB workspace</td>
</tr>
<tr>
<td>DataSource2</td>
<td>Name of the array containing the Compare To signal data for signals imported from the MATLAB workspace</td>
</tr>
<tr>
<td>LinkToPlot (default)</td>
<td>Link to a plot of each comparison result</td>
</tr>
</tbody>
</table>

Example: 'ColumnsToReport',metadata

**ShortenBlockPath — Block path length**

*true (default) | false*

Block path length.

- **true** -- Report shortened block path length.
- **false** -- Include the full block path in the report.
Example: 'ShortenBlockPath',false

**LaunchReport — Open report when created**

true (default) | false

Open report when created.

- true -- Open the report when it is created.
- false -- Do not open the report upon creation.

Example: 'LaunchReport',false

**SignalsToReport — Signals to include in a Compare Runs report**

'REportOnlyMismatchedSignals' (default) | 'ReportAllSignals'

Signals to include in a Compare Runs report.

- ReportOnlyMismatchedSignals -- Include only signals that fall out of tolerance.
- ReportAllSignals -- Include all signals.

Example: 'SignalsToReport','ReportAllSignals'

**See Also**

Simulink.sdi.Signal | Simulink.sdi.compareRuns | Simulink.sdi.compareSignals | Simulink.sdi.createRun

**Topics**

“Inspect and Compare Data Programmatically”
“Create Plots Using the Simulation Data Inspector”
“Save and Share Simulation Data Inspector Data and Views”

**Introduced in R2011b**
Simulink.sdi.resetRunNamingRule

**Package:** Simulink.sdi

Revert the Simulation Data Inspector run naming rule to default

**Syntax**

Simulink.sdi.resetRunNamingRule

**Description**

Simulink.sdi.resetRunNamingRule resets the run naming rule the Simulation Data Inspector uses to assign a name to runs created through simulating a Simulink model to its default 'Run <run_index>: <model_name>'.

**Examples**

**Modify then Reset Run Naming Rule**

This example shows how to use the Simulation Data Inspector API to modify the Simulation Data Inspector run naming rule, check a run's name, reset the run naming rule to its default, and check the run naming rule.

```matlab
% Load model
load_system('sldemo_fuelsys')

% Modify run naming rule
Simulink.sdi.setRunNamingRule('<model_name> Run <run_index>')

% Simulate system
sim('sldemo_fuelsys')

% Check run name
runIDs = Simulink.sdi.getAllRunIDs;
```
runID = runIDs(end);
run = Simulink.sdi.getRun(runID);
run.name

ans =
'sldemo_fuelsys Run 1'

% Reset the run naming rule
Simulink.sdi.resetRunNamingRule

% Check run naming rule
Simulink.sdi.getRunNamingRule

ans =
'Run <run_index>: <model_name>'

Alternatives

You can reset the run naming rule to its default using the Simulation Data Inspector UI. Use the Restore Defaults button on the New Run tab under Simulation Data Inspector Preferences.

Simulink.sdi.clearPreferences restores the run naming rule, along with all other Simulation Data Inspector preferences.

See Also

Simulink.sdi.clearPreferences | Simulink.sdi.getRunNamingRule | Simulink.sdi.setRunNamingRule

Topics

“Inspect and Compare Data Programmatically”
“Organize Your Simulation Data Inspector Workspace”

Introduced in R2015a
Simulink.sdi.save

**Package:** Simulink.sdi

Save Simulation Data Inspector session

**Syntax**

```matlab
Simulink.sdi.save(fileName)
```

**Description**

`Simulink.sdi.save(fileName)` saves all runs, signals, and plot settings as a Simulation Data Inspector session in the file `fileName`.

**Examples**

**Save a Simulation Data Inspector Session**

This example shows how to create, save, and load a Simulation Data Inspector session. The example uses data logging to populate the Simulation Data Inspector with data and then uses the Simulation Data Inspector's programmatic interface to create plots to visualize the data. After saving the data and visualization settings in a session, the Simulation Data Inspector repository is emptied in order to demonstrate how to load the session.

**Create Simulation Data**

This example logs the Stick, alpha, rad, and q, rad/sec signals to generate simulation data using the model `slexAircraftExample` and creates two runs. The first uses a sine input, and the second has a square wave input.

```matlab
% Ensure you start with an empty Simulation Data Inspector repository
Simulink.sdi.clear
```
% Load system
load_system('slexAircraftExample')

% Configure signals to log
Simulink.sdi.markSignalForStreaming('slexAircraftExample/Pilot',1,'on')
Simulink.sdi.markSignalForStreaming('slexAircraftExample/Aircraft Dynamics Model',3,'on')
Simulink.sdi.markSignalForStreaming('slexAircraftExample/Aircraft Dynamics Model',4,'on')

% Change Pilot signal to sine
set_param('slexAircraftExample/Pilot','WaveForm','sine')

% Simulate model
sim('slexAircraftExample')

% Change Pilot signal to square
set_param('slexAircraftExample/Pilot','WaveForm','square')

% Simulate model
sim('slexAircraftExample')

**Access Simulation Data**

Use the Simulation Data Inspector programmatic interface to access the simulation data so you can create plots to visualize the signals.

% Get run objects
runIDs = Simulink.sdi.getAllRunIDs;
sineRunID = runIDs(end-1);
squareRunID = runIDs(end);
sineRun = Simulink.sdi.getRun(sineRunID);
squareRun = Simulink.sdi.getRun(squareRunID);

% Get signal objects
sineOut = sineRun.getSignalByIndex(1);
sineIn = sineRun.getSignalByIndex(3);
squareOut = squareRun.getSignalByIndex(1);
squareIn = squareRun.getSignalByIndex(3);

**Create Plots in the Simulation Data Inspector**

Use the programmatic interface to visualize the signal data from the two simulation runs. You can set the plot layout and plot signals on specific subplots.
% Set subplot layout
Simulink.sdi.setSubPlotLayout(2,1)

% Plot sine data on top plot
sineIn.plotOnSubPlot(1,1,true)
sineOut.plotOnSubPlot(1,1,true)

% Plot square wave data on bottom plot
squareIn.plotOnSubPlot(2,1,true)
squareOut.plotOnSubPlot(2,1,true)

**Save a Simulation Data Inspector Session**

First, view the plots you just created. Then, save the Simulation Data Inspector session as an MLDATX-file to recover your data along with your preference selections and plots.

% View the visualized data in the Simulation Data Inspector
Simulink.sdi.view

% Save the Simulation Data Inspector session
Simulink.sdi.save('myData.mldatx')

**Load a Simulation Data Inspector Session**

First, clear the Simulation Data Inspector repository with `Simulink.sdi.clear` and reset visualization settings with `Simulink.sdi.clearPreferences`. Then, you can load the session to see how the data and settings are preserved.

% Clear Simulation Data Inspector repository and preferences
Simulink.sdi.clear
Simulink.sdi.clearPreferences

% Load session file to view data
Simulink.sdi.load('myData.mldatx');

**Input Arguments**

`fileName` — File name  
character vector

Name for the session file.

Example: 'myData.mldatx'
See Also
Simulink.sdi.close | Simulink.sdi.load

Topics
“Inspect and Compare Data Programmatically”
“Save and Share Simulation Data Inspector Data and Views”

Introduced in R2011b
Simulink.sdi.sendWorkerRunToClient

**Package:** Simulink.sdi

Send run created on parallel workers to the Simulation Data Inspector

**Syntax**

Simulink.sdi.sendWorkerRunToClient
Simulink.sdi.sendWorkerRunToClient(run)

**Description**

Simulink.sdi.sendWorkerRunToClient sends the run most recently generated by the worker to the client MATLAB and imports the run to the Simulation Data Inspector.

Simulink.sdi.sendWorkerRunToClient(run) sends the run corresponding to run to the client MATLAB and imports the run to the Simulation Data Inspector.

**Examples**

**Manually Send Runs from Parallel Workers to the Simulation Data Inspector**

This example shows how to use Simulink.sdi.sendWorkerRunToClient to send runs created using parallel workers manually to the Simulation Data Inspector.

**Setup**

This example runs several simulations of the vdp model, varying the value of the gain, Mu. To set up for the parallel simulation, define a vector of Mu values and configure the Simulation Data Inspector for manual Parallel Computing Toolbox support.

```matlab
% Enable manual Parallel Computing Toolbox support
Simulink.sdi.enablePCTSupport('manual');
```
% Choose several Mu values
MuVals = [1 2 3 4];

**Initialize Parallel Workers**

Use `parpool` to start a pool of four parallel workers. This example calls `parpool` inside an if statement so you only create a parallel pool if you don't already have one. You can use `spmd` to run initialization code common to all workers. For example, load the `vdp` model and select signals to log to runs that we can send to the Simulation Data Inspector on the client MATLAB. To avoid data concurrency issues when simulating with `sim` in `parfor`, create a temporary directory on each worker. After the simulations complete, another `spmd` block deletes the temporary directories.

```matlab
p = gcp('nocreate');
if isempty(p)
    parpool(4);
end
Starting parallel pool (parpool) using the 'local' profile ... connected to 4 workers.
```

```
spmd

    % Load system and select signals to log
    load_system('vdp')
    Simulink.sdi.markSignalForStreaming('vdp/x1',1,'on')
    Simulink.sdi.markSignalForStreaming('vdp/x2',1,'on')

    % Create temporary directory for simulation on worker
    workDir = pwd;
    addpath(workDir)
    tempDir = tempname;
    mkdir(tempDir)
    cd(tempDir)

end

**Run Parallel Simulations with parfor**

To stream data from parallel workers to the Simulation Data Inspector, you have to run parallel simulations using `parfor`. Each worker runs a `vdp` simulation with a different
value of \( \mu \). Simulink cannot access the contents of the `parfor` loop, so the variable `MuVal` is defined in the worker's workspace, where the `vdp` model can see it, using `assignin`.

```
parfor (index = 1:4)
    assignin('base','MuVal',MuVals(index));
    set_param('vdp/Mu','Gain','MuVal')
    sim('vdp')
```

**Access Data and Send Run to Client MATLAB**

You can use the Simulation Data Inspector programmatic interface on the worker the same way you would in the client MATLAB. This example creates a `Simulink.sdi.Run` object and attaches the value of \( \mu \) used in the simulation with the Tag property.

```
IDs = Simulink.sdi.getAllRunIDs;
lastIndex = length(IDs);
runID = Simulink.sdi.getRunIDByIndex(lastIndex);
parRun = Simulink.sdi.getRun(runID);
parRun.Tag = strcat('Mu = ',num2str(MuVals(index)));

Simulink.sdi.sendWorkerRunToClient
```

**Close Temporary Directories and View Runs in the Simulation Data Inspector**

Use another `spmd` section to delete the temporary directories created on the workers once the simulations complete. In each simulation, `Simulink.sdi.sendWorkerRunToClient` imported runs from all the workers into the Simulation Data Inspector. You can view the data and check the run properties to see the value of \( \mu \) used during simulation.

```
spmd

    Simulink.sdi.sendWorkerRunToClient

```

2-957
cd(workDir)
rmdir(tempDir, 's')
rmpath(workDir)
end

Simulink.sdi.view

### Input Arguments

**run** — Run ID or Simulink.sdi.Run object  
runID | Simulink.sdi.Run object

Run ID or Simulink.sdi.Run object corresponding to the run you want to import into the Simulation Data Inspector.

### See Also

Simulink.sdi.WorkerRun | Simulink.sdi.cleanupWorkerResources | Simulink.sdi.enablePCTSupport | Simulink.sdi.isPCTSupportEnabled

### Topics

“Inspect and Compare Data Programmatically”

### Introduced in R2018a
Simulink.sdi.setArchiveRunLimit

**Package:** Simulink.sdi

Specify a limit for the number of runs stored in the Simulation Data Inspector archive

**Syntax**

Simulink.sdi.setArchiveRunLimit(limit)

**Description**

`Simulink.sdi.setArchiveRunLimit(limit)` sets the Simulation Data Inspector archive limit as specified by `limit`. When the number of runs in the archive reaches the limit, the Simulation Data Inspector starts to delete runs from the archive on a first-in, first-out basis. When another run enters the archive, the Simulation Data Inspector deletes the run that has been in the archive the longest. A limit of -1 indicates that the archive has no limit for the number of runs it stores. A limit of 0 means that the archive cannot contain any runs.

**Tip** To retain data for only the current run, configure the Simulation Data Inspector to automatically archive runs and set the archive run limit to 0.

**Examples**

**Configure the Simulation Data Inspector to Retain Only the Current Run**

You can configure the Simulation Data Inspector to retain only the logged data for your current simulation. In iterative design and debugging workflows, this configuration helps prevent accumulation of unwanted logged data on disk. First, check the configuration of the Simulation Data Inspector archive. You can save the parameters to restore your preferences after you finish designing or debugging.
limit = Simulink.sdi.getArchiveRunLimit;
mode = Simulink.sdi.getAutoArchiveMode;

Set the archive limit to 0 and configure the Simulation Data Inspector to automatically archive simulation runs.

setArchiveRunLimit(0)
setAutoArchiveMode(true)

When you simulate your model, the Simulation Data Inspector deletes the previous run and updates the view to show signals in the current simulation.

When you finish designing or debugging your model, you can restore the Simulation Data Inspector archive back to its previous configuration.

Simulink.sdi.setArchiveRunLimit(limit)
Simulink.sdi.setAutoArchiveMode(mode)

**Input Arguments**

*limit — Maximum number of runs to store in the Simulation Data Inspector archive*

scalar

Limit for the number of runs stored in the Simulation Data Inspector archive. When the number of runs in the archive reaches the limit, the Simulation Data Inspector starts to delete runs from the archive, on a first-in, first-out basis. A limit of -1 indicates that there is no limit on the runs stored in the archive. A limit of 0 indicates that the archive cannot contain any runs.

Example: -1
Example: 10

**See Also**

Simulink.sdi.getArchiveRunLimit | Simulink.sdi.getAutoArchiveMode | Simulink.sdi.setAutoArchiveMode

**Topics**

“Iterate Model Design Using the Simulation Data Inspector”
Introduced in R2018b
Simulink.sdi.setAutoArchiveMode

**Package:** Simulink.sdi

Specify whether the Simulation Data Inspector automatically archives simulation runs

**Syntax**

Simulink.sdi.setAutoArchiveMode(mode)

**Description**

Simulink.sdi.setAutoArchiveMode(mode) configures the automatic archive behavior of the Simulation Data Inspector, according to mode. The Simulation Data Inspector automatically archives simulation runs when you specify mode as true. When you specify mode as false, the Simulation Data Inspector does not automatically archive.

**Examples**

**Configure the Simulation Data Inspector Archive**

You can configure the Simulation Data Inspector to retain only the logged data for your current simulation. In iterative design and debugging workflows, this configuration helps prevent accumulation of unwanted logged data on disk. First, check the configuration of the Simulation Data Inspector archive. You can save the parameters to restore your preferences after you finish designing or debugging.

```matlab
limit = Simulink.sdi.getArchiveRunLimit;
mode = Simulink.sdi.getAutoArchiveMode;
```

Set the archive limit to 0 and configure the Simulation Data Inspector to automatically archive simulation runs.

```matlab
Simulink.sdi.setArchiveRunLimit(0)
Simulink.sdi.setAutoArchiveMode(true)
```
When you simulate your model, the Simulation Data Inspector deletes the previous run and updates the view to show signals in the current simulation.

When you finish designing or debugging your model, you can restore the Simulation Data Inspector archive back to its previous configuration.

Simulink.sdi.setArchiveRunLimit(limit)
Simulink.sdi.setAutoArchiveMode(mode)

### Input Arguments

**mode** — Logical specification of Simulation Data Inspector automatic archive behavior

true (default) | false

Logical value that specifies whether the Simulation Data Inspector automatically archives simulation runs.

- true — The Simulation Data Inspector automatically archives simulation runs.
- false — The Simulation Data Inspector does not automatically archive simulation runs.

Example: false

Data Types: logical

### See Also

Simulink.sdi.getArchiveRunLimit | Simulink.sdi.getAutoArchiveMode | Simulink.sdi.setArchiveRunLimit

### Topics

“Iterate Model Design Using the Simulation Data Inspector”

Introduced in R2018b
Simulink.sdi.setBorderOn

Display or hide border on time plots

Syntax

Simulink.sdi.setBorderOn(border)

Description

Simulink.sdi.setBorderOn(border) displays or hides the border on time plots in the Simulation Data Inspector according to the logical input, border. The function sets the value of the Show border setting in the Simulation Data Inspector Time Plot Settings. The setting applies to all time plots in the layout. By default, the Simulation Data Inspector shows the border on time plots.

Examples

Configure Time Plot Border Display

You can use the Simulink.sdi.setBorderOn function to show or hide the border on time plots in the Simulation Data Inspector. By default, the Simulation Data Inspector shows the border on time plots.

This example starts by showing how to use the Simulink.sdi.setBorderOn function to determine whether the border is currently displayed. Subsequent sections show the code to hide and show the border. To show the result, the example generates an image using the Simulink.sdi.snapshot function with settings specified by a Simulink.sdi.CustomSnapshot object.

snapSettings = Simulink.sdi.CustomSnapshot;
snapSettings.Width = 300;
snapSettings.Height = 300;
Get Current Border Display Setting

Before modifying the border display setting for time plots, you can save the current setting to a variable in the workspace in case you want to restore the preference later.

```matlab
border = Simulink.sdi.getBorderOn;
```

Hide the Border on Time Plots

```matlab
Simulink.sdi.setBorderOn(false)
Simulink.sdi.snapshot('from','custom','settings',snapSettings);
```

Show the Border on Time Plots

```matlab
Simulink.sdi.setBorderOn(true)
Simulink.sdi.snapshot('from','custom','settings',snapSettings);
```
Input Arguments

**border — Display or hide time plot border**

true or 1 (default) | false or 0

Specify whether time plot borders are shown or hidden in the Simulation Data Inspector.

- true or 1 — Display border for time plots.
- false or 0 — Hide border for time plots.
See Also
Simulink.sdi.getBorderOn | Simulink.sdi.setGridOn |
Simulink.sdi.setMarkersOn | Simulink.sdi.setSubPlotLayout |
Simulink.sdi.setTickLabelsDisplay | Simulink.sdi.setTicksPosition

Topics
“Create Plots Using the Simulation Data Inspector”

Introduced in R2019b
**Simulink.sdi.setGridOn**

**Package:** Simulink.sdi

Configure grid lines for time plots

**Syntax**

```
Simulink.sdi.setGridOn(grid)
```

**Description**

`Simulink.sdi.setGridOn(grid)` configures the grid lines for time plots in the Simulation Data Inspector according to the input, `grid`. The function configures the Horizontal and Vertical options for the Time Plot Settings in the Simulation Data Inspector. The settings apply to the session and configure the grids for all time plots. By default, the Simulation Data Inspector shows horizontal and vertical grid lines.

**Examples**

**Configure Grid for Time Plots in the Simulation Data Inspector**

You can use `Simulink.sdi.setGridOn` to configure the appearance of the grid for time plots in the Simulation Data Inspector. By default, the Simulation Data Inspector shows horizontal and vertical grid lines.

This example starts by showing how to use `Simulink.sdi.getGridOn` to access the current grid configuration. The subsequent sections show the code for each configuration option for time plot grid in the Simulation Data Inspector. To see the result, the example uses a `Simulink.sdi.CustomSnapshot` object to specify settings for the output of the `Simulink.sdi.snapshot` function.

```
snapSettings = Simulink.sdi.CustomSnapshot;
snapSettings.Width = 300;
snapSettings.Height = 300;
```
**Get Initial Grid Setting**

Before modifying the grid display preferences, you can save the current configuration to a variable in the workspace in case you want to restore the preferences later.

```
gridInit = Simulink.sdi.getGridOn;
```

**Configure Horizontal Grid Lines**

Show only horizontal grid lines in the Simulation Data Inspector.

```
Simulink.sdi.setGridOn('horizontal')
Simulink.sdi.snapshot('from','custom','settings',snapSettings);
```

**Configure Vertical Grid Lines**

Show only vertical grid lines in the Simulation Data Inspector.
Simulink.sdi.setGridOn('vertical')
Simulink.sdi.snapshot('from','custom','settings',snapSettings);

Configure No Grid Lines

Show no grid lines in the Simulation Data Inspector.

Simulink.sdi.setGridOn('off')
Simulink.sdi.snapshot('from','custom','settings',snapSettings);
Configure Horizontal and Vertical Grid Lines

Show horizontal and vertical grid lines in the Simulation Data Inspector.

```matlab
Simulink.sdi.setGridOn('on')
Simulink.sdi.snapshot('from','custom','settings',snapSettings);
```
**Input Arguments**

`grid — Specify visible grid lines`

'on' (default) | 'horizontal' | 'vertical' | 'off'

Specify grid appearance for time plots in the Simulation Data Inspector:

- 'on' shows horizontal and vertical grid lines on time plots in the Simulation Data Inspector.
- 'horizontal' shows only horizontal grid lines on time plots in the Simulation Data Inspector.
- 'vertical' shows only vertical grid lines on time plots in the Simulation Data Inspector.
• 'off' shows no grid lines on time plots in the Simulation Data Inspector.

Compatibility Considerations

Specifying the input as a logical value is not recommended

Starting in R2019b, the Simulink.sdi.setGridOn input values changed to:

• 'on'
• 'horizontal'
• 'vertical'
• 'off'

In R2019a, the Simulink.sdi.setGridOn accepted a logical input to show or hide the grid on time plots in the Simulation Data Inspector:

• true or 1 shows horizontal and vertical grid lines on time plots in the Simulation Data Inspector.
• false or 0 hides horizontal and vertical grid lines on time plots in the Simulation Data Inspector.

Starting in R2019b, use the 'on' input value for the same behavior as the true or 1 input, and use the 'off' input value for the same behavior as the false or 0 input.

See Also
Simulation Data Inspector | Simulink.sdi.clearPreferences |
Simulink.sdi.getGridOn | Simulink.sdi.setBorderOn |
Simulink.sdi.setMarkersOn | Simulink.sdi.setSubPlotLayout |
Simulink.sdi.setTickLabelsDisplay | Simulink.sdi.setTicksPosition

Topics
“Create Plots Using the Simulation Data Inspector”
“View Data with the Simulation Data Inspector”
Introduced in R2019a
Simulink.sdi.setMarkersOn

Package: Simulink.sdi

Show or hide markers for plotted signals

Syntax

Simulink.sdi.setMarkersOn(value)

Description

Simulink.sdi.setMarkersOn(value) shows or hides markers on signals plotted in the Simulation Data Inspector according to the logical input, value. The function configures the Show markers setting in the Simulation Data Inspector. The setting applies to the session and shows or hides markers for all signals plotted in the Simulation Data Inspector. By default, markers are hidden in the Simulation Data Inspector.

Examples

Display Markers in the Simulation Data Inspector

Simulink.sdi.setMarkersOn(true);

Input Arguments

value — Logical input
false or 0 (default) | true or 1

Logical specification of whether markers are shown on plots in the Simulation Data Inspector:

- true or 1 shows markers on plots in the Simulation Data Inspector.
• false or 0 does not show markers on plots in the Simulation Data Inspector.

Data Types: logical

See Also
Simulation Data Inspector | Simulink.sdi.clearPreferences | Simulink.sdi.getMarkersOn | Simulink.sdi.setBorderOn | Simulink.sdi.setGridOn | Simulink.sdi.setSubPlotLayout | Simulink.sdi.setTickLabelsDisplay | Simulink.sdi.setTicksPosition

Topics
“Inspect and Compare Data Programmatically”

Introduced in R2017b
Simulink.sdi.setRunNamingRule

Package: Simulink.sdi

Specify the Simulation Data Inspector run naming rule

Syntax

Simulink.sdi.setRunNamingRule('rule')

Description

Simulink.sdi.setRunNamingRule('rule') sets the Simulation Data Inspector rule for naming runs created by simulating a Simulink model.

Examples

Modify Run Naming Rule Then Restore Default

This example shows how to use the Simulation Data Inspector API to modify the Simulation Data Inspector run naming rule, check a run's name, restore default preferences, and check the run naming rule.

% Load model
load_system('sldemo_fuelsys')

% Modify run naming rule
Simulink.sdi.setRunNamingRule('<model_name> Run <run_index>')

% Simulate system
sim('sldemo_fuelsys')

% Check run name
runIDs = Simulink.sdi.getAllRunIDs;
runID = runIDs(end);
fuelRun = Simulink.sdi.getRun(runID);
fuelRun.name

ans =
'sldemo_fuelsys Run 1'

% Clear preferences to reset the run naming rule
Simulink.sdi.clearPreferences

% Check run naming rule
Simulink.sdi.getRunNamingRule

ans =
'Run <run_index>: <model_name>'

**Input Arguments**

'rule' — Simulation Data Inspector run naming rule

'Run <run_index>: <model_name> (default) | character vector

Simulation Data Inspector run naming rule for runs created by simulating a Simulink model. The character vector specifying the run naming rule can include plain text and any of the following tokens that represent data pulled from each run:

- `<run_index>` - Run's index in the Simulation Data Inspector repository.
- `<model_name>` - Name of the model simulated to create the run.
- `<time_stamp>` - Start time for the simulation that created the run.
- `<sim_mode>` - Simulation mode used for the simulation that created the run.

Example: '<time_stamp> Simulation <run_index>: <model_name>'
Example: '<model_name> - <run_index>'

**Alternatives**

You can modify the run naming rule using the Simulation Data Inspector UI in the **Preferences** menu. You can rename a run by modifying the **Name** property of its **Simulink.sdi.Run** object.
See Also
Simulink.sdi.Run | Simulink.sdi.clearPreferences |
Simulink.sdi.getRunNamingRule | Simulink.sdi.resetRunNamingRule

Topics
“Inspect and Compare Data Programatically”
“Organize Your Simulation Data Inspector Workspace”

Introduced in R2011b
Simulink.sdi.setRunOverwrite

**Package:** Simulink.sdi

(Removed) Enable and disable run overwrite mode for a Simulation Data Inspector run

**Note** The Simulink.sdi.setRunOverwrite function has been removed. You can configure the Overwrite Run mode behavior using the Simulink.sdi.setAutoArchiveMode and Simulink.sdi.setArchiveRunLimit functions. For further details, see “Compatibility Considerations”.

**Syntax**

Simulink.sdi.setRunOverwrite(runID,overwrite)

**Description**

Simulink.sdi.setRunOverwrite(runID,overwrite) configures run overwrite mode for the run corresponding to runID, according to the value specified for overwrite.

**Examples**

**Configure the Simulation Data Inspector Archive**

You can configure the Simulation Data Inspector to retain only the logged data for your current simulation. In iterative design and debugging workflows, this configuration helps prevent accumulation of unwanted logged data on disk. First, check the configuration of the Simulation Data Inspector archive. You can save the parameters to restore your preferences after you finish designing or debugging.

```matlab
limit = Simulink.sdi.getArchiveRunLimit;
mode = Simulink.sdi.getAutoArchiveMode;
```
Set the archive limit to 0 and configure the Simulation Data Inspector to automatically archive simulation runs.

```matlab
Simulink.sdi.setArchiveRunLimit(0)
Simulink.sdi.setAutoArchiveMode(true)
```

When you simulate your model, the Simulation Data Inspector deletes the previous run and updates the view to show signals in the current simulation.

When you finish designing or debugging your model, you can restore the Simulation Data Inspector archive back to its previous configuration.

```matlab
Simulink.sdi.setArchiveRunLimit(limit)
Simulink.sdi.setAutoArchiveMode(mode)
```

**Input Arguments**

- **runID** — Numeric run identifier
  
  scalar

  Run ID for the run you want to set run overwrite for. The Simulation Data Inspector assigns run IDs when it creates runs. You can get the run ID for a run using `Simulink.sdi.getAllRunIDs` or `Simulink.sdi.getRunIDByIndex`.

- **overwrite** — Run overwrite enable/disable
  
  true | false

  Logical value for run overwrite mode. Specify `overwrite` as `true` to enable run overwrite mode for a run, and set `overwrite` to `false` to disable run overwrite mode for a run.

**Compatibility Considerations**

*Simulink.sdi.setRunOverwrite function has been removed*

*Errors starting in R2018b*

The `Simulink.sdi.setRunOverwrite` function is removed in R2018b with the introduction of the Simulation Data Inspector archive. You can configure the Overwrite
Run mode behavior using the new `Simulink.sdi.setAutoArchiveMode` and `Simulink.sdi.setArchiveRunLimit` functions. Enable the Simulation Data Inspector to automatically archive simulation runs, and set the archive limit to 0.

```matlab
Simulink.sdi.setAutoArchiveMode(true);
Simulink.sdi.setArchiveRunLimit(0);
```

With these settings, the Simulation Data Inspector retains the data for the current simulation run and discards the data for the previous simulation run. The view in the Simulation Data Inspector also updates to show signals in the current run, with consistent line styles and colors.

You can disable each aspect of the behavior separately. To retain prior run data, set the archive limit to a nonzero value. If you want no limit on the number of runs stored in the archive, you can specify `-1` as the archive limit. When you disable the **Automatically archive** setting, the Simulation Data Inspector does not move the prior run to the archive or update the view to show signals from the current run.

**See Also**

- `Simulink.sdi.getArchiveRunLimit`
- `Simulink.sdi.setAutoArchiveMode`
- `Simulink.sdi.setArchiveRunLimit`
- `Simulink.sdi.setAutoArchiveMode`

**Topics**

- “Iterate Model Design Using the Simulation Data Inspector”

**Introduced in R2011b**
Simulink.sdi.setSubPlotLayout

**Package:** Simulink.sdi

Set subplot layout in the Simulation Data Inspector

**Syntax**

Simulink.sdi.setSubPlotLayout(r,c)

**Description**

Simulink.sdi.setSubPlotLayout(r,c) sets the grid layout of plots in the Simulation Data Inspector using the specified number of rows, r, and columns, c.

**Examples**

**Change Subplot Layout**

% Change subplot layout to 4 rows and 2 columns
Simulink.sdi.setSubPlotLayout(4,2);

**Create a Run and View the Data**

This example shows how to create a run, add data to it, and then view the data in the Simulation Data Inspector.

**Create Data for the Run**

This example creates timeseries objects for a sine and a cosine. To visualize your data, the Simulation Data Inspector requires at least a time vector that corresponds with your data.

% Generate timeseries data
time = linspace(0, 20, 100);
sine_vals = sin(2*pi/5*time);
sine_ts = timeseries(sine_vals, time);
sine_ts.Name = 'Sine, T=5';

cos_vals = cos(2*pi/8*time);
cos_ts = timeseries(cos_vals, time);
cos_ts.Name = 'Cosine, T=8';

Create a Simulation Data Inspector Run and Add Your Data

To give the Simulation Data Inspector access to your data, use the create method and create a run. This example modifies some of the run's properties to help identify the data. You can easily view run and signal properties with the Simulation Data Inspector.

% Create a run
sinusoidsRun = Simulink.sdi.Run.create;
sinusoidsRun.Name = 'Sinusoids';
sinusoidsRun.Description = 'Sine and cosine signals with different frequencies';

% Add timeseries data to run
sinusoidsRun.add('vars', sine_ts, cos_ts);

Plot Your Data Using the Simulink.sdi.Signal Object

The getSignalByIndex method returns a Simulink.sdi.Signal object that can be used to plot the signal in the Simulation Data Inspector. You can also programmatically control aspects of the plot's appearance, such as the color and style of the line representing the signal. This example customizes the subplot layout and signal characteristics.

% Get signal, modify its properties, and change Checked property to true
sine_sig = sinusoidsRun.getSignalByIndex(1);
sine_sig.LineColor = [0 0 1];
sine_sig.LineDashed = '-.';
sine_sig.Checked = true;

% Add another subplot for the cosine signal
Simulink.sdi.setSubPlotLayout(2, 1);

% Plot the cosine signal and customize its appearance
cos_sig = sinusoidsRun.getSignalByIndex(2);
cos_sig.LineColor = [0 1 0];
cos_sig.plotOnSubPlot(2, 1, true);
% View the signal in the Simulation Data Inspector
Simulink.sdi.view

Close the Simulation Data Inspector and Save Your Data
Simulink.sdi.close('sinusoids.mat')

Input Arguments

\( r \) — Number of rows
integer

Number of rows in the subplot grid layout, specified as a whole number between 1 and 8, inclusive.
Example: 2
Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | char

\( c \) — Number of columns
integer

Number of columns in the subplot grid layout, specified as a whole number between 1 and 8, inclusive.
Example: 2
Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | char

See Also
Simulink.sdi.Signal.plotOnSubPlot | Simulink.sdi.clearAllSubPlots | Simulink.sdi.clearPreferences | Simulink.sdi.setBorderOn | Simulink.sdi.setGridOn | Simulink.sdi.setMarkersOn | Simulink.sdi.setTickLabelsDisplay | Simulink.sdi.setTicksPosition

Topics
“Inspect and Compare Data Programmatically”
“Create Plots Using the Simulation Data Inspector”

Introduced in R2016a
Simulink.sdi.setTableGrouping

Package: Simulink.sdi

Change signal grouping hierarchy in Inspect pane

Syntax

Simulink.sdi.setTableGrouping
Simulink.sdi.setTableGrouping('group1')
Simulink.sdi.setTableGrouping('group1', 'group2')
Simulink.sdi.setTableGrouping('group1', 'group2', 'group3')

Description

Simulink.sdi.setTableGrouping groups signals in the Inspect pane of the Simulation Data Inspector as a flat list.

Simulink.sdi.setTableGrouping('group1') groups signals in the Inspect pane by the parameter specified by group1.

Simulink.sdi.setTableGrouping('group1', 'group2') groups signals in the Inspect pane first by the parameter specified by group1 and then by the parameter specified by group2.

Simulink.sdi.setTableGrouping('group1', 'group2', 'group3') groups signals in the Inspect pane first by the parameter specified by group1, then by the parameter specified by group2, and finally by the parameter specified by group3. If you have a Simscape license, you have three options for table grouping.

Examples
Group Data by Model and Then Data Hierarchy

You can group your data in the navigation pane to visualize subsystem and data hierarchy relationships among signals clearly.

Simulink.sdi.setTableGrouping('Subsystems','DataHierarchy');

Input Arguments

'group1' — Signal grouping parameter
character vector

Parameter used to group signals in the Inspect pane of the Simulation Data Inspector. You can specify two grouping parameters, or three if you have a Simscape license.

- DataHierarchy groups signals according to any data hierarchy in the model, for example grouping signals in a bus together.
- SubSystems groups signals according to the model's subsystem hierarchy.
- PhysmodHierarchy groups signals according to the Simscape block structure. This parameter is only available with a Simscape license.

Example: 'SubSystems'
Example: 'DataHierarchy'
Data Types: char

See Also
Simulink.sdi.clearPreferences | Simulink.sdi.view

Topics
“Inspect and Compare Data Programmatically”
“Organize Your Simulation Data Inspector Workspace”

Introduced in R2016a
Simulink.sdi.setTickLabelsDisplay

Configure tick label visibility for time plot axes

Syntax

Simulink.sdi.setTickLabelsDisplay(label)

Description

Simulink.sdi.setTickLabelsDisplay(label) displays labels for the tick marks on axes of time plots in the Simulation Data Inspector according to the input, label. The function sets the value of the **Tick labels** setting in the Simulation Data Inspector **Time Plot Settings**. The setting applies to all time plots in the layout. By default, the Simulation Data Inspector displays tick mark labels for both time plot axes.

Examples

Configure Tick Mark Label Visibility for Time Plots

You can use the Simulink.sdi.setTickLabelsDisplay function to configure the visibility of tick mark labels for the t- and y-axes on time plots in the Simulation Data Inspector. By default, the Simulation Data Inspector displays tick marks for both axes.

This example starts by showing how to use the Simulink.sdi.getTickLabelsDisplay function to access the current tick mark label visibility. Subsequent sections show the code to specify each available option for tick mark label visibility on time plots in the Simulation Data Inspector. To see the result of each configuration, the example generates an image using the Simulink.sdi.snapshot function with settings specified by a Simulink.sdi.CustomSnapshot object.

```matlab
snapSettings = Simulink.sdi.CustomSnapshot;
snapSettings.Width = 300;
snapSettings.Height = 300;
```
Get Current Tick Mark Label Visibility

Before modifying the visibility of the tick mark labels on time plots, you can save the current configuration to a variable in the workspace in case you want to restore the preference later.

\[
\text{initTickLabels} = \text{Simulink.sdi.getTickLabelsDisplay};
\]

Show Only t-Axis Tick Mark Labels

Show the tick mark labels for the \( t \)-Axis.

\[
\text{Simulink.sdi.setTickLabelsDisplay('t-axis')}\
\text{Simulink.sdi.snapshot('from','custom','settings',snapSettings)};
\]
**Show Only y-Axis Tick Mark Labels**

Show the tick mark labels for the y-axis.

```matlab
Simulink.sdi.setTickLabelsDisplay('y-axis')
Simulink.sdi.snapshot('from','custom','settings',snapSettings);
```

**Hide Tick Mark Labels**

You can hide the tick mark labels for both axes.

```matlab
Simulink.sdi.setTickLabelsDisplay('none')
Simulink.sdi.snapshot('from','custom','settings',snapSettings);
```
Show All Tick Mark Labels

By default, the Simulation Data Inspector shows tick mark labels for both axes on time plots.

Simulink.sdi.setTickLabelsDisplay('all')
Simulink.sdi.snapshot('from','custom','settings',snapSettings);
Input Arguments

**label — Tick mark labels to display**

'\textit{all}' (default) | '\textit{t-axis}' | '\textit{y-axis}' | '\textit{none}'

Specify tick mark labels to display on time plots in the Simulation Data Inspector.

- '\textit{all}' — Display tick mark labels for both time plot axes.
- '\textit{t-axis}' — Display tick mark labels only on the $t$-axis for time plots.
- '\textit{y-axis}' — Display tick mark labels only on the $y$-axis for time plots.
- '\textit{none}' — Do not display tick mark labels.
See Also
Simulink.sdi.clearPreferences | Simulink.sdi.getTickLabelsDisplay |
Simulink.sdi.setBorderOn | Simulink.sdi.setGridOn |
Simulink.sdi.setMarkersOn | Simulink.sdi.setSubPlotLayout |
Simulink.sdi.setTicksPosition

Topics
“Create Plots Using the Simulation Data Inspector”

Introduced in R2019b
Simulink.sdi.setTicksPosition

Configure position for tick marks on time plots

Syntax

Simulink.sdi.setTicksPosition(position)

Description

Simulink.sdi.setTicksPosition(position) sets the location of tick marks on time plots in the Simulation Data Inspector according to the input, position. The function configures the Ticks setting in the Simulation Data Inspector Time Plot Settings. The setting applies to all time plots in the layout. By default, the Simulation Data Inspector displays tick marks on the outside of the plot area.

Examples

Configure Position for Tick Marks on Time Plots

You can use Simulink.sdi.setTicksPosition function to specify the position for tick marks on time plots in the Simulation Data Inspector. By default, the Simulation Data Inspector displays tick marks outside of the plot area for time plots.

This example starts by showing how to use the Simulink.sdi.getTicksPosition function to access the current position of tick marks. Subsequent sections show the code to specify each configuration option for the position of tick marks on time plots in the Simulation Data Inspector. To see the result, the example generates an image using Simulink.sdi.snapshot with settings specified by a Simulink.sdi.CustomSnapshot object.

snapSettings = Simulink.sdi.CustomSnapshot;
snapSettings.Width = 300;
snapSettings.Height = 300;
Get Current Tick Mark Position

Before modifying the position of tick marks on time plots, you can save the current configuration to a variable in the workspace in case you want to restore the preference later.

```
initTickPos = Simulink.sdi.getTicksPosition;
```

Position Tick Marks Inside Plot Area

Position the tick marks inside the plot area to increase the amount of space used by the plot area.

```
Simulink.sdi.setTicksPosition('inside')
Simulink.sdi.snapshot('from','custom','settings',snapSettings);
```
**Hide Tick Marks on Time Plots**

You can hide the tick marks for time plots in the Simulation Data Inspector. Hiding the tick marks expands the plot area.

```matlab
Simulink.sdi.setTicksPosition('none')
Simulink.sdi.snapshot('from','custom','settings',snapSettings);
```

**Position Tick Marks Outside Plot Area**

By default, the Simulation Data Inspector displays tick marks outside of the plot area for time plots.

```matlab
Simulink.sdi.setTicksPosition('outside')
Simulink.sdi.snapshot('from','custom','settings',snapSettings);
```
Input Arguments

position — Position for tick marks on time plots
'outside' (default) | 'inside' | 'none'

Specify the position for tick marks on time plots in the Simulation Data Inspector.

Tip Move tick marks to the inside of time plots to increase the area available to the plot area.

Example: 'inside'
See Also
Simulink.sdi.clearPreferences | Simulink.sdi.getTicksPosition |
Simulink.sdi.setBorderOn | Simulink.sdi.setGridOn |
Simulink.sdi.setMarkersOn | Simulink.sdi.setSubPlotLayout |
Simulink.sdi.setTickLabelsDisplay

Topics
“Create Plots Using the Simulation Data Inspector”

Introduced in R2019b
Simulink.sdi.snapshot

Package: Simulink.sdi

Capture contents of Simulation Data Inspector plots

Syntax

\[
\text{fig} = \text{Simulink.sdi.snapshot} \\
[\text{fig}, \text{image}] = \text{Simulink.sdi.snapshot} \\
\text{Simulink.sdi.snapshot(Name, Value)} \\
\text{Simulink.sdi.snapshot(Name, Value)} \\
\text{Simulink.sdi.snapshot(Name, Value)}
\]

Description

\[
\text{fig} = \text{Simulink.sdi.snapshot} \text{ creates a figure of the plotting area in the open Simulation Data Inspector session with the figure handle fig.}
\]

\[
[\text{fig}, \text{image}] = \text{Simulink.sdi.snapshot} \text{ creates a figure of the plotting area in the open Simulation Data Inspector session with the figure handle fig and returns the image data in the array, image.}
\]

\[
\text{Simulink.sdi.snapshot(Name, Value)} \text{ captures an image of the Simulation Data Inspector plots according to the options specified by name-value pairs.}
\]

\[
\text{fig} = \text{Simulink.sdi.snapshot(Name, Value)} \text{ captures an image of the Simulation Data Inspector plots according to the options specified by name-value pairs. This syntax returns the figure handle, fig, if a figure is created.}
\]

\[
[\text{fig}, \text{image}] = \text{Simulink.sdi.snapshot(Name, Value)} \text{ captures an image of the Simulation Data Inspector plots according to the options specified by name-value pairs. This syntax returns the figure handle, fig, and an array of image data, image, when appropriate for the specified options.}
\]
Examples

Copy View Settings to a Run

This example shows how to copy view settings from one run to another and how to create figures using the Simulink.sdi.CustomSnapshot object.

Simulate Your Model and Get a Run Object

Configure the vdp model to save output data, and run a simulation to create data.

```matlab
load_system('vdp')
set_param('vdp','SaveFormat','Dataset','SaveOutput','on')
set_param('vdp/Mu','Gain','1');
sim('vdp');
```

Use the Simulation Data Inspector programmatic interface to access the run data.

```matlab
runIndex = Simulink.sdi.getRunCount;
runID = Simulink.sdi.getRunIDByIndex(runIndex);
vdpRun = Simulink.sdi.getRun(runID);
```

Modify Signal View Settings

Use the Simulink.sdi.Run object to access signals in the run. Then, modify the signal view settings. This example specifies the line color and style for each signal. The view settings for the run comprise the view settings for each signal and view settings specified for the plot area.

```matlab
sig1 = vdpRun.getSignalByIndex(1);
sig2 = vdpRun.getSignalByIndex(2);

sig1.LineColor = [0 0 1];
sig1.LineDashed = '-.';

sig2.LineColor = [1 0 0];
sig2.LineDashed = ':';
```

Capture a Snapshot from the Simulation Data Inspector

Create a Simulink.sdi.CustomSnapshot object and use the Simulink.sdi.snapshot function to programmatically capture a snapshot of the contents of the Simulation Data Inspector.
snap = Simulink.sdi.CustomSnapshot;

You can use properties of the Simulink.sdi.CustomSnapshot object to configure the plot settings, like the subplot layout and axis limits, and to plot signals. When you use a Simulink.sdi.CustomSnapshot object to create your figure, these plot settings do not affect the Simulation Data Inspector.

snap.Rows = 2;
snap.YRange = {{-2.25 2.25},[-3 3]};
snap.plotOnSubPlot(1,1,sig1,true)
snap.plotOnSubPlot(2,1,sig2,true)

Use Simulink.sdi.snapshot to generate the figure you specified in the properties of the Simulink.sdi.CustomSnapshot object.

fig = Simulink.sdi.snapshot(“from”,”custom”,”to”,”figure”,”settings”,snap);
Copy the View Settings to a New Simulation Run

Simulate the model again, with a different \( \mu \) value. Use the Simulation Data Inspector programmatic interface to access the simulation data.

```matlab
set_param('vdp/Mu','Gain','5')
sim('vdp')
```

```matlabunIndex2 = Simulink.sdi.getRunCount;
runID2 = Simulink.sdi.getRunIDByIndex(runIndex2);
run2 = Simulink.sdi.getRun(runID2);
```

To create a plot of the new output data that looks like the one you created in the previous step, you can copy the view settings to the run in a single line of code using
Simulink.sdi.copyRunViewSettings. The Simulink.sdi.copyRunViewSettings function does not automatically update plot settings in Simulink.sdi.CustomSnapshot objects, so specify the plot input as false.

sigIDs = Simulink.sdi.copyRunViewSettings(runID,runID2,false);

**Capture a Snapshot of the New Simulation Run**

Use the Simulink.sdi.CustomSnapshot object to capture a snapshot of the new simulation run. First, clear the signals from the subplots. Then, plot the signals from the new run and capture another snapshot.

snap.clearSignals
snap.YRange = {{-2.25 2.25],[-8 8}};
snap.plotOnSubPlot(1,1,sigIDs(1),true)
snap.plotOnSubPlot(2,1,sigIDs(2),true)

fig = snap.snapshot("to","figure");
Input Arguments

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name, Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside quotes. You can specify several name and value pair arguments in any order as Name1, Value1, ..., NameN, ValueN.

Example: 'to', 'figure', 'props', {'Name', 'My Data'}
**from — Content to include in the snapshot**

'opened' (default) | 'active' | 'comparison' | 'custom'

Content to include in the snapshot.

- 'opened' — Include all subplots in the graphical viewing area of the open Simulation Data Inspector session.
- 'active' — Include only the active (selected) subplot in the open Simulation Data Inspector session.
- 'comparison' — Include the comparison plots for the selected comparison run or signal in the open Simulation Data Inspector session.
- 'custom' — Include contents specified by the settings name-value pair Simulink.sdi.CustomSnapshot object. You can use the 'from','custom' option to create a snapshot without opening the Simulation Data Inspector or affecting your open Simulation Data Inspector session. Include a settings name-value pair when you specify 'from','custom'.

Example: 'from','comparison'

Data Types: char | string

**to — Type of snapshot to create**

'image' (default) | 'figure' | 'file' | 'clipboard'

Type of snapshot to create.

- 'image' — Create a figure containing an image of the specified Simulation Data Inspector contents and return the figure handle and an array of image data. When you specify 'to','image', the fig and image outputs both have value.
- 'figure' — Replicate the specified Simulation Data Inspector contents in a MATLAB figure and return the figure handle. When you specify 'to','figure' the fig output has value, and the image output is empty.
- 'file' — Save a snapshot of the specified Simulation Data Inspector contents to a PNG file with the name specified by the filename name-value pair. If you do not specify a filename name-value pair, the file is named plots.png. When you specify 'to','file', the fig and image outputs are both empty.
- 'clipboard' — Copy the plots to your system clipboard. From the clipboard, you can paste the image into another program such as Microsoft Word. When you specify 'to','clipboard', the fig and image outputs are both empty.

Example: 'to','file'
Data Types: char | string

filename — Name for image file
'plots.png' (default) | character array | string

Name of the image file to store the snapshot when you specify 'to','file'.
Example: 'filename','MyImage.png'
Data Types: char | string

props — Properties to customize the figure
cell array

Figure properties, specified as a cell array. You can include settings for the figure properties described in Figure Properties.
Example: 'props',{ 'Name','MyData' , 'NumberTitle' , 'off' }
Data Types: char | string

settings — Custom snapshot settings
Simulink.sdi.CustomSnapshot

Simulink.sdi.CustomSnapshot object specifying custom snapshot settings. You can use the settings name-value pair to specify the dimensions of the image in pixels, subplot layout, and limits for the x- and y-axes.
Example: 'settings',customSnap
Data Types: char | string

Output Arguments

fig — Figure handle
figure handle

Handle for the figure. When a figure is not created with your specified options, the fig output is empty.

image — Image data
array
Array of image data. The image output has value when you use Simulink.sdi.snapshot without any input arguments or without a to name-value pair and when you specify 'to','image'.

See Also
Simulink.sdi.CustomSnapshot | Simulink.sdi.Signal | Simulink.sdi.clear | Simulink.sdi.clearPreferences | Simulink.sdi.setMarkersOn | Simulink.sdi.setSubPlotLayout | Simulink.sdi.view

Topics
“Inspect and Compare Data Programmatically”
“Create Plots Using the Simulation Data Inspector”

Introduced in R2018a
Simulink.sdi.view

Package: Simulink.sdi

Open the Simulation Data Inspector

Syntax

Simulink.sdi.view

Description

Simulink.sdi.view opens the Simulation Data Inspector. You can write a script to plot your data and customize the Simulation Data Inspector properties and then use Simulink.sdi.view to see the results.

Examples

Open the Simulation Data Inspector from the Command Line

You can open the Simulation Data Inspector from the MATLAB command line to visualize, inspect, and analyze your data.

Simulink.sdi.view

Create a Run and View the Data

This example shows how to create a run, add data to it, and then view the data in the Simulation Data Inspector.
Create Data for the Run

This example creates timeseries objects for a sine and a cosine. To visualize your data, the Simulation Data Inspector requires at least a time vector that corresponds with your data.

```matlab
% Generate timeseries data
time = linspace(0, 20, 100);
sine_vals = sin(2*pi/5*time);
sine_ts = timeseries(sine_vals, time);
sine_ts.Name = 'Sine, T=5';

cos_vals = cos(2*pi/8*time);
cos_ts = timeseries(cos_vals, time);
cos_ts.Name = 'Cosine, T=8';
```

Create a Simulation Data Inspector Run and Add Your Data

To give the Simulation Data Inspector access to your data, use the create method and create a run. This example modifies some of the run's properties to help identify the data. You can easily view run and signal properties with the Simulation Data Inspector.

```matlab
% Create a run
sinusoidsRun = Simulink.sdi.Run.create;
sinusoidsRun.Name = 'Sinusoids';
sinusoidsRun.Description = 'Sine and cosine signals with different frequencies';

% Add timeseries data to run
sinusoidsRun.add('vars', sine_ts, cos_ts);
```

Plot Your Data Using the Simulink.sdi.Signal Object

The getSignalByIndex method returns a Simulink.sdi.Signal object that can be used to plot the signal in the Simulation Data Inspector. You can also programmatically control aspects of the plot's appearance, such as the color and style of the line representing the signal. This example customizes the subplot layout and signal characteristics.

```matlab
% Get signal, modify its properties, and change Checked property to true
sine_sig = sinusoidsRun.getSignalByIndex(1);
sine_sig.LineColor = [0 0 1];
sine_sig.LineDashed = '-.';
sine_sig.Checked = true;
```
% Add another subplot for the cosine signal
Simulink.sdi.setSubPlotLayout(2, 1);

% Plot the cosine signal and customize its appearance
cos_sig = sinusoidsRun.getSignalByIndex(2);
cos_sig.LineColor = [0 1 0];
cos_sig.plotOnSubPlot(2, 1, true);

% View the signal in the Simulation Data Inspector
Simulink.sdi.view

Close the Simulation Data Inspector and Save Your Data
Simulink.sdi.close('sinusoids.mat')

Alternatives
You can open the Simulation Data Inspector from the Simulink Editor toolbar with the
Simulation Data Inspector button.

See Also
Simulink.sdi.clear | Simulink.sdi.clearPreferences | Simulink.sdi.close
| Simulink.sdi.setSubPlotLayout

Topics
“Inspect and Compare Data Programmatically”
“View Data with the Simulation Data Inspector”

Introduced in R2011b
Simulink.SimulationData.createStructOfTimeseries

Create structure of timeseries data to load as simulation input for bus

Syntax

tsStruct = Simulink.SimulationData.createStructOfTimeseries(busObj, tsStructIn)
tsStruct = Simulink.SimulationData.createStructOfTimeseries(busObj, tsCellArray)
tsStructArray = Simulink.SimulationData.createStructOfTimeseries(busObj, tsCellArray, dims)
tsStruct = Simulink.SimulationData.createStructOfTimeseries(tsArray)

Description

tsStruct = Simulink.SimulationData.createStructOfTimeseries(busObj, tsStructIn) creates a structure with attributes that match those specified by the Simulink.Bus object, busObj, and data specified by the structure of timeseries objects, tsStructIn.

When the names in the input structure do not match the names specified by the bus object, the function renames the fields in the output structure to match the bus object specification. When other attributes such as data type and complexity do not match, the function returns an error.

You can use this syntax to create a simulation input that fully or partially specifies the data for a bus. You can also use this syntax to rename the structure elements to match the names in the Simulink.Bus object.

tsStruct = Simulink.SimulationData.createStructOfTimeseries(busObj, tsCellArray) creates a structure of timeseries objects with attributes that match those specified by the Simulink.Bus object, busObj, and data specified by the cell array of timeseries objects, tsCellArray.
When the names in the input structure do not match the names specified by the bus object, the function renames the fields in the output structure to match the bus object specification. When other attributes such as data type and complexity do not match, the function returns an error.

You can use this syntax to create simulation input that fully or partially specifies the data for a bus using a flat list of timeseries objects. The function maps the timeseries objects to the hierarchy specified by the Simulink.Bus object using a depth-first search.

```matlab
tsStructArray = Simulink.SimulationData.createStructOfTimeseries(busObj,tsCellArray,dims)
```
does an array of timeseries structures where the attributes of each structure match those defined by the Simulink.Bus object, `busObj` with the data specified by the cell array of timeseries objects, `tsCellArray`. The input `dims` specifies the dimensions of the array.

```matlab
tsStruct = Simulink.SimulationData.createStructOfTimeseries(tsArray)
```
does a structure of timeseries objects from the bus data stored in the Simulink.TsArray object, `tsArray`. In versions before R2016a, signal logging creates Simulink.TsArray objects to store logged bus data. Use this syntax when you want to use bus data logged in a release before R2016a using ModelDataLogs format as simulation input.

## Examples

### Use Logged Bus Data as Simulation Input

You can use the `Simulink.SimulationData.createStructOfTimeseries` function to create structures of timeseries objects to use as simulation input for bus signals. This example shows you how to create a structure of timeseries to load into a model using timeseries data logged from a different simulation.

### Create a Structure of Timeseries

Open the `ex_log_structtimeseries` model.

```matlab
open_system('ex_log_structtimeseries')
```
The model uses Constant blocks and Bus Creator blocks to build two buses, `bus1` and `bus2`, with signals `a`, `b`, `c`, and `d`. The model uses signal logging to log the bus data. Create bus data by simulating the model.

```matlab
out = sim('ex_log_structtimeseries');
```


```matlab
logsout = out.logsout
```

```matlab
logsout =
Simulink.SimulationData.Dataset 'logsout' with 2 elements
```

<table>
<thead>
<tr>
<th>Name</th>
<th>BlockPath</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>[1x1 Signal]  <code>bus1</code>  <code>ex_log_structtimeseries/Bus Creator</code></td>
</tr>
<tr>
<td>2</td>
<td>[1x1 Signal]  <code>bus2</code>  <code>ex_log_structtimeseries/Bus Creator1</code></td>
</tr>
</tbody>
</table>
- Use braces {} to access, modify, or add elements using index.

You can use the get function to select the Simulink.SimulationData.Signal object for bus2. The bus data is in the Values property of the Simulink.SimulationData.Signal object. The data representing bus2 is logged in a structure containing timeseries objects named c and d.

```matlab
logout.get(2).Values
```

```matlab
ans =
    struct with fields:
        c: [1x1 timeseries]
        d: [1x1 timeseries]
```

**Loading Model Configuration**

Open the `ex_load_structtimeseries` model, which uses the logged simulation data as input.

```matlab
open_system('ex_load_structtimeseries')
```

![Diagram of the model](image)

The model uses the InBus Import block to load input bus data. A Bus Selector block chooses signals from the bus to display on Display blocks.

Double-click the InBus block and check its **Data type** on the **Signal Attributes** tab of the dialog. The data type is specified by a Simulink.Bus object called `bus`. 
Close the dialog and open the Model Explorer. On the **Callbacks** tab, you can see the model uses its `PreLoadFcn` to define the `Simulink.Bus` object that defines the data type for the Inport block.

Open the Configuration Parameters and view the specification for the **Input** parameter on the **Data Import/Export** pane. The model uses the variable `inputBus` for its **Input**.

**Create Simulation Input from the Structure of timeseries Data**

To load the data logged for `bus1`, you only need to assign the structure data to the **Input** variable for the model.

```matlab
inputBus = logsout.get(1).Values
```

```matlab
inputBus =

    struct with fields:
        a: [1x1 timeseries]
        b: [1x1 timeseries]
```

When you simulate the model, the Display blocks show the values 1 and 2 logged in `bus1` and loaded into the model.

To load the data logged for `bus2`, you need to use the `Simulink.Bus` object that defines the Inport block data type and `Simulink.SimulationData.createStructOfTimeseries` to create a structure of timeseries with names specified by the `Simulink.Bus` object.

```matlab
inputBus = Simulink.SimulationData.createStructOfTimeseries('bus',...
  logsout.get(2).Values)
```
inputBus =

    struct with fields:
        a: [1x1 timeseries]
        b: [1x1 timeseries]

When you simulate the model, the Display blocks show the values 3 and 4 logged in bus2 and loaded into the model.

**Partially Specify Simulation Input for a Bus**

This example shows how to use the `Simulink.SimulationData.createStructOfTimeseries` function to create partially specified simulation input for a bus. This example logs data from `ex_log_structtimeseries` and then loads that data into `ex_load_structtimeseries`.

**Create timeseries Data**

First, open and simulate the `ex_log_structtimeseries` model. The model logs two bus signals, bus1 and bus2, created using Constant blocks and Bus Creator blocks. Access the `logssout Dataset` in the `Simulink.SimulationOutput` object, out.

```matlab
open_system('ex_log_structtimeseries')
out = sim('ex_log_structtimeseries');

logssout = out.logssout;
```
You can use a structure of `timeseries` data or a cell array of `timeseries` data to partially specify simulation input for a bus.

**Partially Specify Bus Data with a Structure of `timeseries` Data**

Open the model `ex_load_structtimeseries` that will load some of the data you logged in the previous section.

```matlab
open_system('ex_load_structtimeseries')
```

Use the `get` function to access the structure of `timeseries` data logged for `bus1`.

```matlab
bus1 = logsout.get(1).Values;
```
Then, replace the \( b \) data with \([]\).

\[
\text{bus1.b} = []; \\
\]

The \texttt{ex_load_structtimeseries} model uses the variable \texttt{inputBus} as its \texttt{Input}. The \texttt{Simulink.Bus} object, \texttt{bus}, that defines the data type for the Inport block is defined in the \texttt{PreLoadFcn} callback for the \texttt{ex_load_structtimeseries} model. Because the signal names in \texttt{bus1} match the \texttt{Simulink.Bus} object specification for the Inport block in the \texttt{ex_load_structtimeseries} model, you can use the logged structure without modification. To load the data for \texttt{bus1}, assign \texttt{bus1} to the variable \texttt{inputBus}.

\[
\text{inputBus} = \text{bus1}; \\
\]

Simulate the model. The Display blocks show the logged value 1 for \( a \) and 0 for \( b \). The simulation uses ground values when you do not specify data for the signal.

\[
\text{loadOut} = \text{sim}('ex_load_structtimeseries'); \\
\]

Now, load the data logged for \texttt{bus2}. The signal names in \texttt{bus2} do not match the \texttt{Simulink.Bus} object specification for the Inport block in the \texttt{ex_load_structtimeseries} model. Modify the data in the structure to partially specify input data for the bus. Then, use the \texttt{Simulink.SimulationData.createStructOfTimeseries} function to change the names in the structure to match the bus specification.

\[
\text{bus2} = \text{logsout.get}(2).\text{Values}; \\
\text{bus2.d} = []; \\
\text{inputBus} = \text{bus2}; \\
\text{inputBus} = \text{Simulink.SimulationData.createStructOfTimeseries}('bus',\text{inputBus}); \\
\]

Simulate the model. The Display blocks show the logged value 3 for \( a \) and 0 for \( b \).

\[
\text{loadOut} = \text{sim}('ex_load_structtimeseries'); \\
\]

**Partially Specify Bus Data with a Cell Array of Timeseries Data**

When you have timeseries data, you can use \texttt{Simulink.SimulationData.createStructOfTimeseries} to partially specify simulation input for a bus using a cell array of the timeseries data. Load the timeseries data for signal \( d \) in \texttt{bus2} as part of a partial bus specification for the Inport block in the \texttt{ex_load_structtimeseries} model. The \texttt{PreLoadFcn} callback for the \texttt{ex_load_structtimeseries} model defines the \texttt{Simulink.Bus} object, \texttt{bus}, that defines the data type for the Inport block.
d = logsout.get(2).Values.d;

inputBus = Simulink.SimulationData.createStructOfTimeseries('bus',
    {d,[]});

Simulate the model. The Display block for signal a in the ex_load_structtimeseries
model shows the data logged in signal d from the ex_log_structtimeseries model.
The Display block for signal b shows 0.

loadOut = sim('ex_load_structtimeseries');

Load Simulation Input Data for an Array of Buses

This example shows how to use the Simulink.SimulationData.createStructOfTimeseries function to generate simulation input for an array of buses. You create timeseries data by simulating one model. Then, you create an input structure using the logged data to load into an array of buses in another model.

Create timeseries Data

To start, open the ex_log_structtimeseries model.

open_system('ex_log_structtimeseries')
The model creates two buses, `bus1` and `bus2`, using Constant blocks and Bus Creator blocks. The signals are named `a`, `b`, `c`, and `d`. Create logged bus data by simulating the model.

```matlab
out = sim('ex_log_structtimeseries');
```

The output `out` contains a `Simulink.SimulationData.Dataset` object, `logsout`, with the logged data. You can access the `bus1` and `bus2` signals using the `get` function. The data for each signal is in the `Simulink.SimulationData.Signal` object `Values` parameter. You can access the bus elements using a dot followed by the signal name. `bus1` is the first signal in the `Dataset` object and contains signals `a` and `b`. `bus2` contains signals `c` and `d`.

```matlab
logsout = out.logsout;

a = logsout.get(1).Values.a;
b = logsout.get(1).Values.b;
c = logsout.get(2).Values.c;
d = logsout.get(2).Values.d;
```
**Loading Model Configuration**

Open the model `ex_structtimeseries_aob`, which uses an array of buses as input.

```matlab
open_system('ex_load_structtimeseries_aob')
```

The model uses the `InAoB` Inport block to load simulation input. Selector blocks select a bus from the array of buses, and Bus Selector blocks select signals to show in the Display blocks.

Double-click the `InAoB` block and look at the **Signal Attributes** pane of the dialog. The **Data type** for the block is set to **Bus** with the type defined by the `Simulink.Bus` object, `bus`. The **Port dimensions** parameter is set to `[2 1]`.

You can see the definition for the `Simulink.Bus` object, `bus`, in the **Callbacks** tab in the Model Explorer. This model uses the `PreLoadFcn` to define the bus object.

Open the Model Configuration Parameters and look at the **Input** parameter. The model uses the variable `inputAoB` as input.

**Create Array of Buses Simulation Input**

Use `Simulink.SimulationData.createStructOfTimeseries` and the data logged in the first section to create a structure to load as simulation input for the array of buses. Specify the dimensions as `[2 1]` to match the dimensions of the `InAoB` block.

```matlab
inputAoB = Simulink.SimulationData.createStructOfTimeseries('bus',... {a,b,c,d},[2 1]);
```
When you simulate the model, the Display blocks show the data for signals a, b, c, and d logged from the ex_log_structtimeseries model. The array of buses contains two buses with signals a and b. Simulink.SimulationData.createStructOfTimeseries renamed signals c and d to match the Simulink.Bus specification used by the array of buses.

inputAoB(2)

ans =

struct with fields:
   a: [1x1 timeseries]
   b: [1x1 timeseries]

Simulate the model. The display blocks show the logged values.

aob_out = sim('ex_load_structtimeseries_aob');

Create a Structure of timeseries Objects from a Simulink.TSArray Object

In releases before R2016a, when you log simulation data using ModelDataLogs format, bus data is stored as a Simulink.TSArray object. You cannot log data using ModelDataLogs format using a release after R2016a. In this example, the logged data, logsout, was logged in ModelDataLogs format using a release before R2016a. The variable logsout contains data for a single bus, bus1.

logsout

logsout =

Simulink.ModelDataLogs (log_modeldatalog):  
   Name                     Elements  Simulink Class
   bus1                      2        TsArray

To load the logged data as simulation input for a bus, create a structure of timeseries objects from the data in bus1.

struct_of_ts = ...
Simulink.SimulationData.createStructOfTimeseries(logsout.bus1)
struct_of_ts =
    const1_sig: [1x1 timeseries]
    const2_sig: [1x1 timeseries]

Input Arguments

busObj — Name of Simulink.Bus object
character vector

Name of the Simulink.Bus object that specifies the attributes for the data in the output structure of timeseries objects. When you want to load the structure of timeseries objects as simulation input for a bus, the busObj is the bus that defines the data type for the root-level Inport block.

Simulink.SimulationData.createStructOfTimeseries validates the input timeseries attributes including data type and complexity against the Simulink.Bus object specification. When element names do not match between the Simulink.Bus specification and the input timeseries data, Simulink.SimulationData.createStructOfTimeseries renames the timeseries data to match the bus specification. When other attributes do not match, the function returns an error.

Example: 'MyInputBus'

tsStructIn — Structure of timeseries objects
structure of timeseries data

Structure of timeseries data for use in creating the output structure of timeseries objects according to the Simulink.Bus object. The structure must have the same hierarchy as the Simulink.Bus object.

To partially specify data for a bus, use [] in the place of the bus element you want to use ground values.

tsCellArray — Cell array of timeseries objects
cell array of timeseries objects

Cell array of timeseries objects specifying the data for the output structure of timeseries objects.
To partially specify data for a bus, use [] in the place of the bus element you want to use ground values.

The `Simulink.SimulationData.createStructOfTimeseries` function maps the `timeseries` elements of the cell array to the hierarchy specified by the `Simulink.Bus` object using a depth-first search.

Example: `{ts1,ts2,ts3}`
Example: `{ts1,[],ts3}`

**Dependencies**

When you specify the `dims` argument, the number of cells in the cell array must match the number of individual signal elements in the `Simulink.Bus` object multiplied by the product of the specified dimensions.

**dims — Dimensions for output array**

vector

Dimensions for the array of `timeseries` structures, specified as a vector.

When you specify the dimensions as a scalar, n, the function creates a 1-by-n array.

Example: `[2,1]`

**Dependencies**

When you specify the `dims` argument, the number of cells in the cell array must match the number of individual signal elements in the `Simulink.Bus` object multiplied by the product of the specified dimensions.

Data Types: double

**tsArray — Simulink.TsArray object**

Simulink.TsArray

`Simulink.TsArray` object.

In versions prior to R2016a, signal logging creates `Simulink.TsArray` objects to store logged bus data. Use this syntax when you want to use data logged using `ModelDataLogs` format in a version before R2016a to create simulation input for a bus.

Example: `myTsArrayObj`
Output Arguments

tsStruct — Structure of timeseries objects
struct

Structure of timeseries objects with attributes specified by the Simulink.TsArray or Simulink.Bus input. You can load the structure of timeseries objects as simulation input for a bus.

tsStructArray — Array of timeseries structures
array

Array of structures of timeseries objects with dimensions specified by the dims input.

See Also
Simulink.Bus | Simulink.TsArray | timeseries

Topics
“Load Data to Root-Level Input Ports”
“Load Bus Data to Root-Level Input Ports”

Introduced in R2013a
getAsDatastore

**Class:** Simulink.SimulationData.DatasetRef  
**Package:** Simulink.SimulationData

Get matlab.io.datastore.SimulationDatastore representation of element from referenced Dataset object

**Syntax**

```verbatim
element = Simulink.SimulationData.DatasetRef.getAsDatastore(datasetref_elements)
```

**Description**

```verbatim
element = Simulink.SimulationData.DatasetRef.getAsDatastore(datasetref_elements) returns a matlab.io.datastore.SimulationDatastore representation of an element or collection of elements from the referenced dataset, based on index, name, or block path of the element.

You can represent a Dataset element as a matlab.io.datastore.SimulationDatastore object if the element was placed into the MAT-file using either of these approaches:

- Log Dataset format data to persistent storage (MAT-file).
- Place the element into a Simulink.SimulationData.Dataset object and saved the Dataset object to a v7.3 MAT-file.

The SimulationDatastore representation for a Dataset element creates a SimulationDatastore object for the Values field of that element. The SimulationDatastore representation supports streaming of the data for the Values property of the element into other simulations or into MATLAB.
```
**Note** You cannot use create a `SimulationDatastore` for `Dataset` elements that contain these types of data:

- Array

You can use `SimulationDatastore` objects to:

- Refer to logged simulation data that is stored on disk in a MAT-file.
- Specify signals to stream incrementally from disk to a simulation.
- Provide a basis for big data analysis using MATLAB functions.

### Input Arguments

**datasetref_element** — Element of referenced dataset in MAT-file  

index of the element

Element of a referenced dataset in a MAT-file, specified as an index, name (as a character vector), or block path (as a character vector).

### Output Arguments

**element** — Element accessed using `SimulationDatastore` object  

`matlab.io.datastore.SimulationDatastore` object | `Simulink.SimulationData.Signal`, `Simulink.SimulationData.State` or similar object, whose Values data uses a `matlab.io.datastore.SimulationDatastore` object

Element accessed using `SimulationDatastore` object, returned as either a `matlab.io.datastore.SimulationDatastore` object or a `Simulink.Signal`, `Simulink.State`, or similar object, whose Values data uses a `matlab.io.datastore.SimulationDatastore` object.

### Examples
Use a SimulationDatastore to Reference a Signal’s Data in a DatasetRef

Log signal data to persistent storage (select the **Log Dataset data to file** configuration parameter) and simulate a model.

Create a DatasetRef for the signal logging Dataset data (logsout) in the out.mat MAT-file.

```matlab
sigLogRef = Simulink.SimulationData.DatasetRef('out.mat','logsout');
firstSig = sigLogRef.getAsDatastore(1)
```

```matlab
count = 0;
while count < 100
    sim('my_model', 'ExternalInput', ds);
end
```

Alternative

To streamline the use of indexing, you can use curly braces ({{}}) syntax to obtain a SimulationDatastore object for DatasetRef object signal values. The requirements and results are the same as using `getAsDatastore` for example, if you log signal data to persistent storage (select the **Log Dataset data to file** configuration parameter) and simulate a model.

```matlab
sigLogRef = Simulink.SimulationData.DatasetRef('out.mat','logsout');
firstSig = sigLogRef{1}
```

```matlab
ans =
```
Simulink.SimulationData.Signal
Package: Simulink.SimulationData

Properties:

Name: 'x1'
PropagatedName: ''
BlockPath: [1x1 Simulink.SimulationData.BlockPath]
PortType: 'outport'
PortIndex: 1
Values: [1×1 matlab.io.datastore.SimulationDatastore]

See Also

Topics
“Load Big Data for Simulations”

Introduced in R2017a
Simulink.SimulationData.DatasetRef.getDatasetVariableNames

List names of Dataset variables in MAT-file

Syntax

varNames = Simulink.SimulationData.DatasetRef.getDatasetVariableNames(matFile)

Description

varNames = Simulink.SimulationData.DatasetRef.getDatasetVariableNames(matFile)
lists the names of variables for Dataset data in a MAT-file.

Examples

List Variable Names in MAT-File

Suppose that you simulate a model using the default variable names for signal logging data and states data. You enable the Configuration Parameters > Data Import/Export > Log Dataset data to file and use the default MAT-file name of out.mat.

List the variable names in the MAT-file.

varNames = Simulink.SimulationData.DatasetRef.getDatasetVariableNames('out.mat')
varNames =
    'xout'    'logsout'

**Tips**

To get the names of Dataset variables in the MAT-file, using the Simulink.SimulationData.DatasetRef.getDatasetVariableNames function processes faster than using the who, or whos functions.

**Input Arguments**

matFile — MAT-file that contains Dataset variables  
character vector  

MAT-file that contains Dataset variables, specified as a character vector. The character vector specifies the path to the MAT-file.

**Output Arguments**

varNames — Names of Dataset variables in MAT-file  
cell array  

Names of Dataset variables in MAT-file, returned as a cell array.

**See Also**


**Topics**

“Log Data to Persistent Storage”
“Load Big Data for Simulations”

**Introduced in R2016a**
Simulink.SimulationData.forEachTimeseries

Call function on each timeseries object

Syntax

dataResults = Simulink.SimulationData.forEachTimeseries(functionHandle,inputData)

Description

dataResults = Simulink.SimulationData.forEachTimeseries(functionHandle,inputData) runs the specified function handle on all MATLAB timeseries objects contained in inputData.

Examples

Find Minimum for Logged Bus Signal Data

This example shows how to use the forEachTimeseries function to run the min function on each timeseries object in the logged data for the COUNTERBUS signal.

Open the model and simulate it.

code:
```matlab
open_system('sldemo_mdlref_bus')
sim('sldemo_mdlref_bus');
```
Access the signal logging data. For this model, that data is stored in the `topOut` variable.

```matlab
topOut
Simulink.SimulationData.Dataset 'topOut' with 4 elements
```

<table>
<thead>
<tr>
<th>Name</th>
<th>BlockPath</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 [1x1 Signal]</td>
<td>COUNTERBUS sldemo.mdlref_bus/Concatenate</td>
</tr>
<tr>
<td>2 [1x1 Signal]</td>
<td>OUTERDATA sldemo.mdlref_bus/CounterA</td>
</tr>
<tr>
<td>3 [1x1 Signal]</td>
<td>INCREMENTBUS sldemo.mdlref_bus/IncrementBusCreator</td>
</tr>
<tr>
<td>4 [1x1 Signal]</td>
<td>INNERDATA ...erA sldemo.mdlref_counter_bus/COUNTER</td>
</tr>
</tbody>
</table>

- Use braces `{ }` to access, modify, or add elements using index.

Find the values for the COUNTERBUS element.

```matlab
counterbusData = topOut{1}.Values
counterbusData =
2x1 struct array with fields:
```
Run the \texttt{min} function on the \texttt{counterbus} data.

\begin{verbatim}
ret = Simulink.SimulationData.forEachTimeseries(@min,counterbusData)
ret =
2x1 struct array with fields:
  data
  limits

Explore the returned data.

ret(1)
ans =
  data: 0
  limits: [1x1 struct]

ret(2).limits
ans =
  upper_saturation_limit: 40
  lower_saturation_limit: 0
\end{verbatim}

**Input Arguments**

\texttt{functionHandle} — Function to run on MATLAB timeseries objects

\begin{verbatim}
function handle
\end{verbatim}

Function to run on \texttt{timeseries} objects, specified as a function handle. For information about specifying function handles, see “Pass Function to Another Function” (MATLAB).

The function that you use with \texttt{forEachTimeseries}:

- Can be either a built-in function or a user-specified function
- Must return a scalar

If the function that you use with \texttt{forEachTimeseries} takes:
• One argument, specify the function handle and the input data. For example:

```matlab
ret = Simulink.SimulationData.forEachTimeseries(@min,data);
```

• More than one argument, specify the function handle as `@(x)` and then specify the function, using `x` as the first argument. For remaining arguments, specify values. For example, this command runs the `resample` function on MATLAB timeseries objects in `data`, for the time vector `[2.5 3]`.

```matlab
ret = Simulink.SimulationData.forEachTimeseries(@(x)... (resample(x,[2.5 3]),data);
```

**inputData — Data to run specified function on**
MATLAB timeseries object | array of timeseries | structure with timeseries at leaf nodes | array of structures with timeseries at leaf nodes

Data to run specified function on, specified as timeseries data.

**Output Arguments**

**dataResults — Data resulting from running specified function**
MATLAB timeseries object | array of timeseries | structure with timeseries at leaf nodes | array of structures with timeseries at leaf nodes

Data resulting from running specified function, returned using the format and hierarchy of the input data.

**Related Links**

MATLAB timeseries “Function Handles”

Introduced in R2016b
Simulink.SimulationData.signalLoggingSelector

Open Signal Logging Selector

Syntax

Simulink.SimulationData.signalLoggingSelector(modelName)

Description

Simulink.SimulationData.signalLoggingSelector(modelName) opens the Signal Logging Selector dialog box for the model that you specify with modelName.

Input Arguments

modelName

Character vector that specifies the name of the model for which you want to open the Signal Logging Selector dialog box.

Example

Open the Signal Logging Selector dialog box for the sldemo_mdlref_bus.mdl.

Simulink.SimulationData.signalLoggingSelector('sldemo_mdlref_bus')

See Also

Simulink.ModelDataLogs | Simulink.SimulationData.Dataset

Topics

“Override Signal Logging Settings”
Introduced in R2011a
**setName**

**Class:** Simulink.SimulationData.Unit  
**Package:** Simulink.SimulationData

Specify name of logging data units

**Syntax**

unitObject = setName(unitObj,unitName)

**Description**

unitObject = setName(unitObj,unitName) sets the name for the Simulink.SimulationData.Unit object to the name specified in unitName.

**Input Arguments**

unitObj — Logging data unit object to name  
Simulink.SimulationData.Unit object

Logging data unit object to name, specified as a Simulink.SimulationData.Unit object.

unitName — Name of logging data unit  
character vector

Name of logging data unit, specified as a character vector.

**Output Arguments**

name — Name of logging data units  
character vector

Name of logging data units, returned as a character vector.
Examples

Name a Logging Data Unit Object

```matlab
inchesUnit = Simulink.SimulationData.Unit('in');
inchesUnit = setName(inchesUnit,'inches')

inchesUnit =
    Units with properties:
        Name: 'inches'
```

See Also

Simulink.SimulationData.Unit

Topics

“Log Signal Data That Uses Units”
“Convert Logged Data to Dataset Format”
“Prepare Model Inputs and Outputs”

Introduced in R2011a
Simulink.SimulationData.updateDatasetFormatLogging

Convert model and its referenced models to use Dataset format for signal logging

Syntax

Simulink.SimulationData.updateDatasetFormatLogging(top_model)
Simulink.SimulationData.updateDatasetFormatLogging(top_model, variants)

Description

**Note** The ModelDataLogs class is supported for backwards compatibility. Starting in R2016a, you cannot log data in the ModelDataLogs format. Signal logging uses the Dataset format. In R2016a or later, when you open a model from an earlier release that had used ModelDataLogs format, the model simulated in use Dataset format. You do not need to use this command to update the signal logging format for a model that uses model referencing. Opening the model in R2016a or later uses Dataset format for all signal logging.

You can convert signal logging data from ModelDataLogs to Dataset format. Converting to Dataset format makes it easier to post-process with other logged data (for example, logged states), which can also use Dataset format. For more information, see “Convert Logged Data to Dataset Format”.

If you have legacy code that uses the ModelDataLogs API, you can encounter situations that require updates to your code or model. See “Migrate Scripts That Use Legacy ModelDataLogs API”.

Simulink.SimulationData.updateDatasetFormatLogging(top_model) converts the top-level model and all of its referenced models to use the Dataset format for signal logging instead of the ModelDataLogs format. You can convert signal logging data from ModelDataLogs to Dataset format. Converting to Dataset format makes it easier to
post-process with other logged data (for example, logged states), which can also use Dataset format. For more information, see “Convert Logged Data to Dataset Format”.

If a Model block has the **Generate preprocessor conditionals** option selected, the function converts all the variants; otherwise, the function converts only the active variant.

`Simulink.SimulationData.updateDatasetFormatLogging(top_model, variants)` specifies which variant models to convert to use the Dataset signal logging format. For details about the `variants` argument, see “Input Arguments” on page 2-1042.

**Input Arguments**

**top_model**
Character vector that specifies the name of the top-level model.

**variants**
Character vector that specifies which variant models to update:
- 'ActivePlusCodeVariants' — (Default) Search all variants if any generate preprocessor conditionals. Otherwise, search only the active variant.
- 'ActiveVariants' — Convert only the active variant.
- 'AllVariants' — Convert all variants.

**More About**

**Dataset**

The **Dataset** format causes Simulink to use a `Simulink.SimulationData.Dataset` object to store the logged signal data. The Dataset format use MATLAB `timeseries` objects to formatting the data.
ModelDataLogs

The ModelDataLogs format causes Simulink to use a Simulink.ModelDataLogs object to store the logged signal data. Simulink.Timeseries and Simulink.TsArray objects provide the format for the data.

Tips

• The conversion function sets the SignalLoggingSaveFormat parameter value to Dataset for all the updated models.
• If you want to save the format updates that the conversion function makes, then ensure that the top-level model, referenced models, and variant models are accessible and writable.
• If a model has no other unsaved changes, the conversion function saves the format updates to the model. If the model has unsaved changes, the function updates the format, but does not save those changes.
• If you use this function for a model that does not include any referenced models, the function converts the top-level model use the Dataset format.

See Also

Simulink.ModelDataLogs | Simulink.ModelDataLogs.convertToDataset | Simulink.SimulationData.Dataset

Topics

“Migrate Scripts That Use Legacy ModelDataLogs API”

Introduced in R2011a
find

Class: Simulink.SimulationOutput
Package: Simulink

Access and display values of simulation results

Syntax

\[ output = \text{simOut}.\text{find}('VarName') \]

Description

\[ output = \text{simOut}.\text{find}('VarName') \] accepts one variable name. Specify \textit{VarName} inside single quotes.

Input Arguments

\begin{itemize}
  \item \textit{VarName}
\end{itemize}

Name of logged variable for which you seek values.

Default:

Output Arguments

\begin{itemize}
  \item \textit{Value}
\end{itemize}

Value of the logged variable name specified in input.
Examples

Simulate \texttt{vdp} and store the values of the variable \texttt{youtNew} in \texttt{yout}.

\begin{verbatim}
simOut = sim('vdp','SimulationMode','rapid','AbsTol','1e-5',...  'SaveState','on','StateSaveName','xoutNew',...  'SaveOutput','on','OutputSaveName','youtNew');
yout = simOut.find('youtNew')
\end{verbatim}

Alternatives

A simpler alternative is to use dot notation. For example, to access data for the \texttt{xoutNew} output variable, you can use this command:

\begin{verbatim}
simOut.xoutNew
\end{verbatim}

Another alternative is to use \texttt{Simulink.SimulationOutput.who} and then \texttt{Simulink.SimulationOutput.get}.

See Also

\texttt{Simulink.SimulationOutput.get} | \texttt{Simulink.SimulationOutput.who}
get

Class: Simulink.SimulationOutput
Package: Simulink

Access and display values of simulation results

Syntax

$output = 
\text{simOut.}
.get('VarName')$

Description

$output = 
\text{simOut.}
.get('VarName')$ accepts one variable name. Specify $VarName$
inside single quotes.

Tip A simpler alternative to using the get function is to use dot notation. For example, to
access data for the $xout$ output variable, you can use this command:

\text{simOut.xout}

Input Arguments

$VarName$

Name of logged variable for which you seek values.

Default:
Output Arguments

Value

Value of the logged variable name specified in input.

Examples

Simulate vdp and store the values of the variable youtNew in yout.

```matlab
simOut = sim('vdp','SimulationMode','rapid','AbsTol','1e-5',...  'SaveState','on','StateSaveName','xoutNew',...  'SaveOutput','on','OutputSaveName','youtNew');
yout = simOut.get('youtNew')
```

Alternatives

A simpler alternative is to use dot notation. For example, to access data for the xout output variable, you can use this command:

```matlab
simOut.xout
```

Another alternative is to use `Simulink.SimulationOutput.who` and then `Simulink.SimulationOutput.find`.

See Also

`Simulink.SimulationOutput.who` | `Simulink.SimulationOutput.find`
getSimulationMetadata

**Class:** Simulink.SimulationOutput  
**Package:** Simulink

Return SimulationMetadata object for simulation

**Syntax**

```matlab
mData = simout.getSimulationMetadata()
```

**Description**

`mData = simout.getSimulationMetadata()` retrieves metadata information in a SimulationMetadata object from the `simout` SimulationOutput object.

**Input Arguments**

- `simout` — Simulation object to get metadata from
  
  Simulation object to get metadata from, specified as a `SimulationOutput` object.

**Output Arguments**

- `mData` — SimulationMetadata object stored in the `simout` SimulationOutput object
  
  SimulationMetadata object stored in the `simout` SimulationOutput object, returned as an object.
Examples

Retrieve Metadata From vdp Simulation

Simulate the vdp model and retrieve metadata information from the simulation.

Simulate the vdp model. Save the results of the Simulink.SimulationOutput object in simout

```matlab
open_system('vdp')
simout = sim(bdroot,'ReturnWorkspaceOutputs','on');
```
Retrieve metadata information about this simulation using `mData`.

```
 mData=simout.getSimulationMetadata()
```

```
 mData =
 SimulationMetadata with properties:
    ModelInfo: [1x1 struct]
    TimingInfo: [1x1 struct]
    ExecutionInfo: [1x1 struct]
    UserString: ''
```
Alternatives

A simpler alternative to use dot notation with the `SimulationMetadata` property. For example:

```plaintext
simOut.SimulationMetadata.ModelInfo
```

Another alternative is to display simulation metadata in the Variable Editor using one of these approaches:

- Select the **Show Simulation Metadata** check box (which displays the data in a tree structure).
- Double-click the **SimulationMetadata** row.
- View the **SimulationMetadata** object.

See Also

- `Simulink.SimulationMetadata`
- `Simulink.SimulationOutput.setUserData`
- `Simulink.SimulationOutput.setUserString`
setUserData

**Class:** Simulink.SimulationOutput  
**Package:** Simulink

Store custom data in `SimulationMetadata` object that `SimulationOutput` object contains

**Syntax**

```matlab
simoutNew = simout.setUserData(CustomData)
```

**Description**

`simoutNew = simout.setUserData(CustomData)` assigns a copy of the `simout` `SimulationOutput` object to `simoutNew`. The copy contains `CustomData` in its `SimulationMetadata` object.

**Input Arguments**

- **`simout`** — *Simulation object to get metadata from*  
  Simulation object to get metadata from, specified as a `SimulationOutput` object.

- **`CustomData`** — *Data to store in a metadata object*  
  Any custom data you want to store in the metadata object.

**Output Arguments**

- **`simoutNew`** — *Simulation object that stores metadata object with custom data*  
  The output `SimulationOutput` object containing `CustomData` in its `SimulationMetadata` object.
A copy of the simout SimulationOutput object that contains CustomData in its SimulationMetadata object, returned as an object.

Examples

Store Data in SimulationMetadata Object of vdp Simulation

Simulate the vdp model. Store custom data in the SimulationMetadata object that the SimulationOutput object contains.

Simulate the vdp model. Save the results of the Simulink.SimulationOutput object in simout.

```matlab
open_system('vdp')
simout=sim(bdroot,'ReturnWorkspaceOutputs','on');
```
Store custom data about the simulation in the `SimulationMetadata` object that `simout` contains.

\[
\text{simout}=\text{simout}.\text{setUserData}(
\text{struct('param1','value1','param2','value2','param3','value3'))}
\]

Use `SimulationOutput.getSimulationMetadata` to retrieve the information you stored.

\[
\text{mData}=\text{simout}.\text{getSimulationMetadata}();
\text{disp(mData.UserData)}
\]

```
    param1: 'value1'
    param2: 'value2'
```
param3: 'value3'

See Also
Simulink.SimulationMetadata|
Simulink.SimulationOutput.getSimulationMetadata|
Simulink.SimulationOutput.setUserString
setUserString

Class: Simulink.SimulationOutput
Package: Simulink

Store custom character vector in SimulationMetadata object that SimulationOutput object contains

Syntax

simoutNew = simout.setUserString(CustomString)

Description

simoutNew = simout.setUserString(CustomString) assigns a copy of the simout SimulationOutput object to simoutNew. The copy contains CustomString in its SimulationMetadata object.

Input Arguments

simout — Simulation object to get metadata from object

Simulation object to get metadata from, specified as a SimulationOutput object.

CustomString — Character vector to store in a metadata object character vector

Any custom character vector you want to store in the metadata object.

Output Arguments

simoutNew — Simulation object that stores metadata object with custom character vector object
A copy of the `simout` SimulationOutput object that contains CustomString in its SimulationMetadata object, returned as an object.

**Examples**

**Store a Character Vector in SimulationMetadata Object of vdp Simulation**

Simulate the `vdp` model. Store a custom character vector in the SimulationMetadata object that the SimulationOutput object contains.

Simulate the `vdp` model. Save the results of the `Simulink.SimulationOutput` object in `simout`.

```matlab
open_system('vdp')
simout = sim(bdroot,'ReturnWorkspaceOutputs','on');
```

---

Copyright 2004-2013 The MathWorks, Inc.
Store a character vector to describe the simulation.

simout=simout.setUserString('First Simulation');

Use SimulationOutput.getSimulationMetadata to retrieve the information you stored.

mData=simout.getSimulationMetadata();
disp(mData.UserString)
First Simulation

**See Also**

`Simulink.SimulationMetadata`
`Simulink.SimulationOutput.getSimulationMetadata`
`Simulink.SimulationOutput.setUserData`
who

Class: Simulink.SimulationOutput  
Package: Simulink

Access and display output variable names of simulation

Syntax

simOutVar = simOut.who

Description

simOutVar = simOut.who returns the names of all simulation output variables, including workspace variables.

Output Arguments

simOutVar

Character vector array of output variable names of simulation.

Examples

Simulate vdp and store the character vector values of the output variable names.

```plaintext
simOut = sim('vdp','SimulationMode','rapid','AbsTol','1e-5',...  
             'SaveState','on','StateSaveName','xoutNew',...  
             'SaveOutput','on','OutputSaveName','youtNew');
simOutVar = simOut.who
```

2-1060
Alternatives

A simpler alternative is to use dot notation. For example, to access data for the `xoutNew` output variable, you can use this command:

`simOut.xoutNew`

See Also

`Simulink.SimulationOutput.find` | `Simulink.SimulationOutput.get`
Simulink.SubSystem.convertToModelReference

Convert subsystem to model reference

Syntax

Simulink.SubSystem.convertToModelReference(gcb,'UseConversionAdvisor',true)

[success,mdlRefBlkHs] = Simulink.SubSystem.convertToModelReference(subsys,mdlRefs)
[success,mdlRefBlkHs] = Simulink.SubSystem.convertToModelReference(subsys,mdlRefs,Name,Value)

Description

Simulink.SubSystem.convertToModelReference(gcb,'UseConversionAdvisor',true) opens the Model Reference Conversion Advisor for the currently selected subsystem block.

[success,mdlRefBlkHs] = Simulink.SubSystem.convertToModelReference(subsys,mdlRefs) converts the specified subsystems to referenced models using the mdlRefs value.

For each subsystem that the function converts, it:

• Creates a model
• Copies the contents of the subsystem into the new model
• Updates any root-level Inport, Outport, Trigger, and Enable blocks and the configuration parameters of the model to match the compiled attributes of the original subsystem
• Copies the contents of the model workspace of the original model to the new model

Before you use the function, load the model containing the subsystem.
[success,mdlRefBlkHs] = Simulink.SubSystem.convertToModelReference(subsys,mdlRefs,Name,Value) uses additional options specified by one or more Name,Value pair arguments.

Examples

Open the Model Reference Conversion Advisor

Open the f14 model.

open_system('f14');

In the f14 model, select the Controller subsystem output signal, click the Simulation Data Inspector button arrow, and select Log Selected Signals.

In the Simulink Editor, select the Controller subsystem. Then open the Model Reference Conversion Advisor from the command line.

Simulink.SubSystem.convertToModelReference(gcb,'UseConversionAdvisor',true);

Perform the conversion using the advisor.

Convert Subsystem to Referenced Model

Convert the Bus Counter subsystem to a referenced model named bus_counter_ref_model.

open_system('sldemo_mdlref_conversion');
Simulink.SubSystem.convertToModelReference(...
    'sldemo_mdlref_conversion/Bus Counter', ...
    'bus_counter_ref_model', ...
    'AutoFix',true,...
    'ReplaceSubsystem',true,...
    'CheckSimulationResults',true);

### Successfully converted Subsystem to Model reference block
Convert Multiple Subsystems to Referenced Models

Convert the two subsystems with one command.

```matlab
open_system('f14');
set_param(gcs,'SaveOutput','on','SaveFormat','Dataset');
set_param(gcs,'SignalResolutionControl','UseLocalSettings');
Simulink.SubSystem.convertToModelReference(...
    {'f14/Controller','f14/Aircraft Dynamics Model'},...
    {'controller_ref_model','aircraft_dynamics_ref_model'},...
    'ReplaceSubsystem',true,...
```
Input Arguments

**subsys** — Subsystems to convert
character vector | subsystem handle | cell array of character vectors | array of subsystem handles

Subsystems to convert, specified as a character vector, subsystem handle, or cell array of strings or array of subsystem handles.

For information on which subsystems you can convert, see “Modify Referenced Models for Conditional Execution”.

Data Types: double

**mdlRefs** — Referenced model names
character vector | cell array of character vectors

Referenced model names, specified as a character vector or cell array of character vectors. Each model name must be 59 characters or less.

If you specify a cell array of subsystems to convert, specify a cell array of referenced model names. Each model name corresponds to the specified subsystem, in the same order.

Name-Value Pair Arguments

Specify optional comma-separated pairs of **Name,Value** arguments. **Name** is the argument name and **Value** is the corresponding value. **Name** must appear inside quotes. You can specify several name and value pair arguments in any order as **Name1,Value1,...,NameN,ValueN**.

Example: `Simulink.SubSystem.convertToModelReference... (engineSubsys,engineModelRef,'ReplaceSubsystem',true)`

**AutoFix** — Fix all conversion issues that can be fixed automatically
false (default) | true

If you set **AutoFix** to **true**, the function fixes all conversion issues that it can fix.
For issues that the function cannot fix, the conversion process generates error messages that you address by modifying the model.

**Note** If you set 'Force' to true, then the function does not automatically fix conversion issues.

Data Types: logical

**Force — Complete conversion even with errors**
false (default) | true

If you set 'Force' to true, the function returns conversion errors as warnings and continues with the conversion without fixing the errors. This option allows you to use the function to do the initial steps of conversion and then complete the conversion process yourself.

If you set Force to true, then the function does not fix conversion issues, even if you set 'AutoFix' to true. However, the success output argument is true, regardless of whether any conversion errors occurred.

**CheckSimulationResults — Compare simulation results before and after conversion**
false (default) | true

Compare simulation results before and after conversion, specified as true or false.

Before performing the conversion, enable signal logging for the subsystem output signals of interest in the model.

For the Simulink.SubSystem.convertToModelReference command, set:

- 'CheckSimulationResults' to true
- 'AbsoluteTolerance'
- 'RelativeTolerance'
- 'SimulationModes' to the same as the simulation mode as in the original model

If the difference between simulation results exceeds the tolerance level, the function displays a message.
**AbsoluteTolerance — Absolute signal tolerance for comparison**

'1e-06' (default) | double

Absolute signal tolerance for comparison, specified as a double. Use the option only if you set `CheckSimulationResults` to `true`.

Data Types: double

**RelativeTolerance — Relative signal tolerance for comparison**

'1e-06' (default) | double

Relative signal tolerance for comparison, specified as a double. Use the option only if you set `CheckSimulationResults` to `true`.

Data Types: double

**DataFileName — Name of file for storing conversion data**

character vector

Name of file for storing conversion data, specified as a character vector. You can specify an absolute or relative path.

You can save the conversion data in a MAT-file (default) or a MATLAB file. If you use a .m file extension, the function serializes all variables to a MATLAB file.

By default, the function uses a file name consisting of the model name plus _conversion_data.mat.

**ReplaceSubsystem — Replace content of each subsystem with Model blocks**

false (default) | true

Replace subsystem blocks with Model blocks, specified as true or false. The Model block references the referenced model.

By default, the function displays the referenced models in separate Simulink Editor windows.

If you set the value to `true`, consider making a backup of the original model before you convert the subsystems. If you want to undo the conversion, having a backup makes it easier to restore the model.

If you set `ReplaceSubsystem` to `true`, the conversion action depends on whether you use the automatic fix options.
• If you use the automatic fixes, then the conversion replaces the Subsystem block with a Model block unless the automatic fixes change the input or output ports. If the ports change, then the conversion includes the contents of the subsystem in a Model block that is inserted in the Subsystem block.
• If you do not use the automatic fixes, then the conversion replaces the Subsystem block with a Model block.

Data Types: logical

CreateWrapperSubsystem — Insert wrapper subsystem to preserve model layout
false (default) | true

Insert wrapper subsystem to preserve model layout, specified as true or false. When you convert a subsystem to a referenced model, you can have the conversion process insert a wrapper subsystem to preserve the layout of a model. The subsystem wrapper contains the Model block from the conversion.

The conversion creates a wrapper subsystem automatically if the conversion modifies the Model block interface by adding ports.

Data Types: logical

SimulationModes — Simulation mode for Model blocks
'Normal' (default) | 'Accelerator'

Simulation mode for Model blocks, specified as a 'Normal' or 'Accelerator'. The simulation mode setting applies to the Model blocks that reference the models that the conversion creates.

BuildTarget — Model reference targets to generate
'Sim' | 'Coder'

Model reference targets to generate.
• 'Sim' — Model reference simulation target
• 'Coder' — Code generation target

Output Arguments

success — Conversion status
1 | 0
Conversion status. A value of 1 indicates a successful conversion.

If you set 'Force' to true, the function returns a value of 1 if the conversion completes. However, the simulation results can differ from the simulation results for the model before conversion.

mdlRefBlkHs — Handles of created Model blocks

`handle of Model block | array of handles of Model blocks`

Handles of created Model blocks, returned as a double or cell array.

Data Types: double

**Tips**

- You cannot convert a parent subsystem and a child of that subsystem at the same time.
- Specifying multiple subsystems to convert with one command can save time, compared to converting each subsystem separately. The multiple subsystem conversion process compiles the model once.
- If you specify multiple subsystems to convert, the conversion process attempts to convert each subsystem. Successfully converted subsystems produce referenced models, even if the conversions of other subsystems fail.
- If you specify multiple subsystems, consider:
  - In the input arguments, setting 'Autofix', 'ReplaceSubsystem', and 'CheckSimulationResults' to true.
  - In the model, setting a short simulation time.
- A data dictionary saves bus objects that it creates as part of the conversion process when both of these conditions exist:
  - The top model uses a data dictionary.
  - All changes to the top model are saved.
- To maintain the behavior of a masked subsystem, perform these additional tasks:
  - For masked callbacks, icons, ports, and documentation, create a backup copy of the masked subsystem before converting it to a referenced model. After conversion, copy the content from the Mask Editor of the backup masked subsystem to the Mask Editor of the new Model block.
• Replace mask parameters with model arguments (see “Parameterize Instances of a Reusable Referenced Model”).

**Note** Referenced models do not support the functionality that you can achieve with mask initialization code to create masked parameters.

### Compatibility Considerations

'BuildTarget' argument value 'RTW' is not recommended

*Not recommended starting in R2019a*

In R2019a, an argument value used with the `Simulink.SubSystem.convertToModelReference` function was updated. The legacy argument value is supported for backward compatibility.

To take advantage of the updated 'BuildTarget' argument value, use 'Coder' instead of 'RTW'.

### See Also

- `Simulink.BlockDiagram.copyContentsToSubsystem`
- `Simulink.Bus.save`
- `Simulink.SubSystem.copyContentsToBlockDiagram`

### Topics

- `sldemo_mdlref_conversion`
- “Convert Subsystems to Referenced Models”
- “Model References”

**Introduced in R2006a**
Simulink.SubSystem.copyContentsToBlockDiagram

Copy contents of subsystem to empty block diagram

Syntax

Simulink.SubSystem.copyContentsToBlockDiagram(subsys, bdiag)

Description

Simulink.SubSystem.copyContentsToBlockDiagram(subsys, bdiag) copies the contents of the subsystem subsys to the block diagram bdiag. The subsystem and block diagram must have already been loaded. The subsystem cannot be part of the block diagram. The function affects only blocks, lines, and annotations; it does not affect nongraphical information such as configuration sets.

This function cannot be used if the destination block diagram contains any blocks or signals. Other types of information can exist in the destination block diagram and are unaffected by the function. Use Simulink.BlockDiagram.deleteContents if necessary to empty the block diagram before using Simulink.SubSystem.copyContentsToBlockDiagram.

Tip  To flatten a model hierarchy by expanding the contents of a subsystem to the system that contains that subsystem, do not use the Simulink.SubSystem.copyContentsToBlockDiagram function. Instead, expand the subsystem, as described in “Expand Subsystem Contents”.

Input Arguments

subsys

Subsystem name or handle
bdiag
Block diagram name or handle

Examples
Copy the graphical contents of f14/Controller, including all nested subsystems, to a new block diagram:

% open f14
open_system('f14');

% create a new model
newbd = new_system;
open_system(newbd);

% copy the subsystem
Simulink.SubSystem.copyContentsToBlockDiagram('f14/Controller', newbd);

% close f14 and the new model
close_system('f14', 0);
close_system(newbd, 0);

See Also
Simulink.BlockDiagram.copyContentsToSubsystem |
Simulink.BlockDiagram.deleteContents |
Simulink.SubSystem.convertToModelReference |
Simulink.SubSystem.deleteContents

Topics
“Interactive Model Editing”
“Create a Subsystem”
“Expand Subsystem Contents”

Introduced in R2007a
**Simulink/SubSystem.deleteContents**

Delete contents of subsystem

**Syntax**

Simulink/SubSystem.deleteContents(subsys)

**Description**

Simulink/SubSystem.deleteContents(subsys) deletes the contents of the subsystem subsys. The function affects only blocks, lines, and annotations. The subsystem must have already been loaded.

**Note** This function does not delete library blocks in a subsystem.

**Input Arguments**

subsys

Subsystem name or handle

**Examples**

Delete the graphical contents of Controller, including all nested subsystems:

load_system('f14');
Simulink/SubSystem.deleteContents('f14/Controller');

**See Also**

Simulink/BlockDiagram.copyContentsToSubsystem
Simulink/BlockDiagram.deleteContents
Simulink.SubSystem.convertToModelReference
Simulink.SubSystem.copyContentsToBlockDiagram

Topics
“Design Model Architecture”
“Create a Subsystem”

Introduced in R2007a
Simulink.SubSystem.getChecksum

Return checksum of nonvirtual subsystem

Syntax

[checksum,details] = Simulink.SubSystem.getChecksum(subsys)

Description

[checksum,details] = Simulink.SubSystem.getChecksum(subsys) returns the checksum of the specified nonvirtual subsystem. Simulink computes the checksum based on the subsystem parameter settings and the blocks the subsystem contains. Virtual subsystems do not have checksums.

One use of this command is to determine why code generated for a subsystem is not being reused.

Note Simulink.SubSystem.getChecksum compiles the model that contains the specified subsystem, if the model is not already in a compiled state. If you need to get the checksum for multiple subsystems and want to avoid multiple compiles, use the command, model([], [], [], 'compile') to place the model in a compiled state before using Simulink.SubSystem.getChecksum.

This command accepts the argument subsys, which is the full name or handle of the nonvirtual subsystem block for which you are returning checksum data.

Examples
Run getChecksum on Model

Run the function Simulink.SubSystem.getChecksum on the model rtwdemo_ssreuse. In the MATLAB editor window, both output structures are displayed. In the workspace pane, double-click on either of the structures to view its contents.

Load the model rtwdemo_ssreuse.

rtwdemo_ssreuse

Select subsystem SS1 and execute the following line of code in the MATLAB editor to get the full name and path to the subsystem SS1:

path_ss1 = gcb

Run the function getChecksum on the subsystem with the following command:

[chksum1, chksum1_details] = Simulink.SubSystem.getChecksum(path_ss1)

The output structures chksum1 and chksum1_details will store the output of the getChecksum function call.

chksum1 =

    struct with fields:
    
        Value: [4×1 uint32]
        MarkedUnique: 0

chksum1_details =

    struct with fields:
    
        ContentsChecksum: [1×1 struct]
        InterfaceChecksum: [1×1 struct]
        ContentsChecksumItems: [359×1 struct]
        InterfaceChecksumItems: [60×1 struct]

Input Arguments

subsys — Name or handle of nonvirtual subsystem
character vector
Input the full name of the nonvirtual subsystem for which you want to calculate the checksum.

Data Types: char

Output Arguments

**checksum** — A structure that stores the value of the checksum and indicates whether subsys contains unique block or subsystem properties which prevent generated code reuse

structure

Checksum information, returned as a structure with the fields:

**Value** — Array of four 32-bit integers that represents the subsystem's 128-bit checksum

4x1 uint32

**MarkedUnique** — True if the subsystem or the blocks it contains have properties that would prevent the code generated for the subsystem from being reused; otherwise, false

bool

**details** — A structure that stores checksum data on model contents and the interface

structure

Checksum information, returned as a structure with the fields:

**ContentsChecksum** — A structure of the same form as checksum, representing a checksum that provides information about all blocks in the system

structure

**InterfaceChecksum** — A structure of the same form as checksum, representing a checksum that provides information about the subsystem's block parameters and connections

structure

**ContentsChecksumItems** — Structure array that Simulink uses to compute the checksum for ContentsChecksum

structure
Structure array returned with the following fields:

Handle — Object for which Simulink added an item to the checksum. For a block, the handle is a full block path. For a block port, the handle is the full block path and a character vector that identifies the port

char array

Identifier — Descriptor of the item Simulink added to the checksum. If the item is a documented parameter, the identifier is the parameter name

char array

Value — Value of the item Simulink added to the checksum. If the item is a parameter, Value is the value returned by get_param(handle, identifier)

type

InterfaceChecksumItems — Structure array that Simulink uses to compute the checksum for InterfaceChecksum structure

Structure array returned with the following fields:

Handle — Object for which Simulink added an item to the checksum. For a block, the handle is a full block path. For a block port, the handle is the full block path and a character vector that identifies the port

char array

Identifier — Descriptor of the item Simulink added to the checksum. If the item is a documented parameter, the identifier is the parameter name

char array

Value — Value of the item Simulink added to the checksum. If the item is a parameter, Value is the value returned by get_param(handle, identifier)

type

See Also
Simulink.BlockDiagram.getChecksum

Introduced in R2006b
Simulink.suppressDiagnostic

Suppress a diagnostic from a specific block

Syntax

Simulink.suppressDiagnostic(source, message_id)
Simulink.suppressDiagnostic(diagnostic)

Description

Simulink.suppressDiagnostic(source, message_id) suppresses all instances of diagnostics represented by message_id thrown by the blocks specified by source.

Simulink.suppressDiagnostic(diagnostic) suppresses the diagnostics associated with MSLDiagnostic object diagnostic.

Examples

Suppress a Warning Thrown By a Block

Using the model from “Suppress Diagnostic Messages Programmatically”, use the Simulink.suppressDiagnostic function to suppress the parameter precision loss warning thrown by the Constant block, one.

Simulink.suppressDiagnostic('Suppressor_CLI_Demo/one',...
     'SimulinkFixedPoint:util:fxpParameterPrecisionLoss');

Input Arguments

source — Block or model object throwing the diagnostic
block path | block handle
The source of the diagnostic, specified as a block path, block handle, cell array of block paths, or cell array of block handles.

To get the block path, use the `gcb` function.

To get the block handle, use the `getSimulinkBlockHandle` function.

Data Types: `char` | `cell`

`message_id` — message identifier of diagnostic
message identifier | cell array of message identifiers

Message identifier of the diagnostic, specified as a character vector or a cell array of character vectors. You can find the message identifier of diagnostics thrown during simulation by accessing the `ExecutionInfo` property of the `Simulink.SimulationMetadata` object associated with a simulation. You can also use the `lastwarn` function.

Data Types: `char` | `cell`

`diagnostic` — Diagnostic object
`MSLDiagnostic` object

Diagnostic specified as an `MSLDiagnostic` object. Access the `MSLDiagnostic` object through the `ExecutionInfo` property of the `Simulink.SimulationMetadata` object.

Data Types: `struct`

**See Also**

`Simulink.SuppressDiagnostic` | `Simulink.SuppressDiagnostic.restore` | `Simulink.getSuppressedDiagnostics` | `Simulink.restoreDiagnostic`

**Topics**

“Suppress Diagnostic Messages Programmatically”
sint

Create Simulink.NumericType object describing signed integer data type

Syntax

a = sint(WordLength)

Description

sint(WordLength) returns a Simulink.NumericType object that describes the data type of a signed integer with a word size given by WordLength.

Note sint is a legacy function. In new code, use fixdt instead. In existing code, replace sint(WordLength) with fixdt(1,WordLength,0).

Examples

Define a 16-bit signed integer data type.

a = sint(16)

a =

NumericType with properties:

    DataTypeMode: 'Fixed-point: binary point scaling'
    Signedness: 'Signed'
    WordLength: 16
    FractionLength: 0
    IsAlias: 0
    DataScope: 'Auto'
    HeaderFile: ''
    Description: ''
See Also
Simulink.NumericType | fixdt | float | sfix | sfrc | ufix | ufrac | uint

Introduced before R2006a
slblocksearchdb.trainfrommodel

Train suggestion engine to improve quick insert results based on one model

Syntax

slblocksearchdb.trainfrommodel(model)

Description

slblocksearchdb.trainfrommodel(model) improves quick insert search results based on a single model.

Examples

Use a Single Model for Training

Train the suggestion engine to use a model to improve results. Assume 'MyModels' is in the current folder. This code uses a relative path.

slblocksearchdb.trainfrommodel('MyModels/model.slx')

Input Arguments

model — Model for training the suggestion engine
model path

Model to train the suggestion engine, specified as a character vector or string scalar of the full model path or relative path. Include the .slx or .mdl extension

Example: 'H:/MyModels/model.slx' 'mymodel.slx'
See Also
slblocksearchdb.trainfrommodelsindir | slblocksearchdb.untrainall | slblocksearchdb.untrainmodel | slblocksearchdb.untrainmodelsindir

Topics
“Improve Quick Block Insert Results”

Introduced in R2018a
slblocksearchdb.trainfrommodelsindir

Train suggestion engine to improve quick insert results based on models in a folder

Syntax

slblocksearchdb.trainfrommodelsindir(folder)
slblocksearchdb.trainfrommodelsindir(folder,'exclude',exclusions)

Description

slblocksearchdb.trainfrommodelsindir(folder) improves search results based on the models in folder, recursively.

slblocksearchdb.trainfrommodelsindir(folder,'exclude',exclusions) excludes the specified models from updating the suggestion engine.

Examples

Train Suggestion Engine and Exclude Folders and Models

Create the cell array exPath for a folder you want to exclude and a model you want to exclude. To train the suggestion engine, use the cell array in the slblocksearchdb.trainfrommodelsindir command.

exPath = {'MyModels/testmodels' 'MyModels/myvdp.slx'}
slblocksearchdb.trainfrommodelsindir('MyModels','exclude',exPath)
The command uses the models in the folder 'MyModels' with the exclusions you specified.

**Input Arguments**

**folder** — Folder whose models to use for training
folder path

Folder whose models to use for training the suggestion engine, specified as an absolute or relative path character vector or string scalar.

Example: 'H:/MyModels/trainingmodels' 'MyModels'

**exclusions** — Models to exclude from training models
cell array of character vectors | string array

Models to exclude from training the suggestion engine, specified as a cell array of character vectors or a string array. Specify folders or models to exclude as a full or relative path. Model names must include the file extension .slx or .mdl.

Example: {'H:/MyModels/trainingmodels' 'MyModels/trainingmodels2' 'MyModels/myvdp.slx'}

**See Also**

slblocksearchdb.trainfrommodel | slblocksearchdb.untrainall |
slblocksearchdb.untrainmodel | slblocksearchdb.untrainmodelsindir

**Topics**

“Improve Quick Block Insert Results”

**Introduced in R2018a**
**slblocksearchdb.untrainall**

Remove the effects of all added models from the suggestion engine

**Syntax**

```
slblocksearchdb.untrainall
```

**Description**

`slblocksearchdb.untrainall` removes the models added to the suggestion engine to improve quick insert results. Use this function when you want to return the database to the default state.

**Examples**

**Add Models and Remove All**

Add some models to the suggestion engine.

```plaintext
slblocksearchdb.trainfrommodelsindir('MyModels')
slblocksearchdb.trainfrommmodel('C:/users/TrainingModels/mymodel.slx')
```

Remove all added models from the suggestion engine.

```plaintext
slblocksearchdb.untrainall
```

**See Also**

`slblocksearchdb.trainfrommodel` | `slblocksearchdb.trainfrommodelsindir` |
`slblocksearchdb.untrainmodel` | `slblocksearchdb.untrainmodelsindir`

**Topics**

“Improve Quick Block Insert Results”
Introduced in R2018a
slblocksearchdb.untrainmodel

Remove the effect of a model from the suggestion engine

Syntax

slblocksearchdb.untrainmodel('model')

Description

slblocksearchdb.untrainmodel('model') removes the effects of a single model from the suggestion engine.

Examples

Remove a Model from the Suggestion Engine

Train the suggestion engine to use a model to improve results.

slblocksearchdb.trainfrommodel('MyModels/model.slx')

Remove the model from the suggestion engine.

slblocksearchdb.untrainmodel('MyModels/model.slx')

Input Arguments

model — Model to remove from the suggestion engine
model path

Model whose effects to remove from the suggestion engine, specified as the full or relative path character vector or string scalar. Include the .slx or .mdl extension.

Example: 'H:/MyModels/model.slx'
See Also
slblocksearchdb.trainfrommodel | slblocksearchdb.trainfrommodelsindir | slblocksearchdb.untrainall | slblocksearchdb.untrainmodelsindir

Topics
“Improve Quick Block Insert Results”

Introduced in R2018a
**slblocksearchdb.untrainmodelsindir**

Remove the effects of models from the suggestion engine

**Syntax**

```matlab
slblocksearchdb.untrainmodelsindir(folder)
slblocksearchdb.untrainmodelsindir(folder,'exclude',exclusions)
```

**Description**

`slblocksearchdb.untrainmodelsindir(folder)` removes models in `folder` from the suggestion engine, recursively.

`slblocksearchdb.untrainmodelsindir(folder,'exclude',exclusions)` excludes the specified models from updating the suggestion engine.

**Examples**

**Remove the Effects of Models from Suggestion Engine**

Create the cell array `exPath` for a folder and a model whose effects you do not want to remove from the suggestion engine. Then use the cell array in the `slblocksearchdb.untrainmodelsindir` command.

```matlab
exPath = {'MyModels/subfolder' 'MyModels/myvdp.slx'}
slblocksearchdb.untrainmodelsindir('MyModels','exclude',exPath)
```
The command removes the models in the folder 'MyModels' with the exclusions you specified.

**Input Arguments**

folder — Folder whose models to remove from the suggestion engine

folder path

Folder whose models to remove from the suggestion engine, specified as an absolute or relative path character vector or string scalar.

Example: 'H:/MyModels/trainingmodels' 'MyModels'

exclusions — Folders and models to exclude from removing

cell array of character vectors | string array

Folders or models to exclude from removing from the suggestion engine, specified as a cell array of character vectors or a string array. Specify the folders or models to exclude as a full or relative path. For models, include the .slx or .mdl extension.

Example: {'H:/MyModels/trainingmodels' 'MyModels/trainingmodels2' 'MyModels/myvdp.slx'}

**See Also**

slblocksearchdb.trainfrommodel | slblocksearchdb.trainfrommodelsindir | slblocksearchdb.untrainall | slblocksearchdb.untrainmodel

**Topics**

“Improve Quick Block Insert Results”

**Introduced in R2018a**
slbuild

Build standalone executable or model reference target for model

Syntax

slbuild(model)
slbuild(model,buildSpec)
slbuild(model,buildSpec,Name,Value)

Description

slbuild(model) builds a standalone Simulink Coder binary executable file from model, using the current model configuration settings. If the model has not been loaded, slbuild loads it before initiating the build process.

Do not use rtwbuild, rtwrebuild, or slbuild commands with parallel language features (Parallel Computing Toolbox), for example, within a parfor or spmd loop. For information about parallel builds of referenced models, see “Reduce Build Time for Referenced Models by Using Parallel Builds” (Simulink Coder).

You cannot use slbuild to build subsystems.

slbuild(model,buildSpec) builds a standalone Simulink Coder binary executable file from model according to the specified code generation action.

slbuild(model,buildSpec,Name,Value) builds a standalone Simulink Coder binary executable file from model as specified by one or more Name,Value pairs.

Examples

Generate Code and Build Executable Image for Model

Generate C code for model rtwdemo_rtwintro.
slbuild('rtwdemo_rtwintro')
% Same operation as ...
% slbuild('rtwdemo_rtwintro','StandaloneCoderTarget')

For the generic real-time (GRT) target, the coder generates these code files and places them in folders rtwdemo_rtwintro_grt_rtw and slprj/grt/_sharedutils.

<table>
<thead>
<tr>
<th>Model Files</th>
<th>Shared Files</th>
<th>Interface Files</th>
<th>Other Files</th>
</tr>
</thead>
<tbody>
<tr>
<td>rtwdemo_rtwintro.c</td>
<td>rtwtypes.h</td>
<td>rtmodel.h</td>
<td>none</td>
</tr>
<tr>
<td>rtwdemo_rtwintro.h</td>
<td>multiword_types.h</td>
<td></td>
<td></td>
</tr>
<tr>
<td>rtwdemo_rtwintro_private.h</td>
<td>builtin typeid types.h</td>
<td></td>
<td></td>
</tr>
<tr>
<td>rtwdemo_rtwintrotypes.h</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

If the following model configuration parameters settings apply, the coder generates additional results.

<table>
<thead>
<tr>
<th>Parameter Setting</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Code Generation &gt; Generate code only</strong> pane is cleared</td>
<td>Executable image rtwdemo_rtwintro.exe</td>
</tr>
<tr>
<td><strong>Code Generation &gt; Report &gt; Create code generation report</strong> is selected</td>
<td>Report appears, providing information and links to generated code files, subsystem and code interface reports, entry-point functions, inports, outports, interface parameters, and data stores</td>
</tr>
</tbody>
</table>

**Clean Top Model Build**

Clean the model build area enough to trigger regeneration of the top model code at the next build.

slbuild('rtwdemo_rtwintro','CleanTopModel')
**Force Top Model Build**

Generate code and build an executable image for `rtwdemo_mdlreftop`, which refers to model `rtwdemo_mdlrefbot`, regardless of model checksums and parameter settings.

```
slbuild('rtwdemo_mdlreftop','StandaloneCoderTarget', ... 'ForceTopModelBuild',true)
```

**Input Arguments**

**model** — Specifies model for the build process

*handle | character vector*

Model for which to build a standalone executable or model reference target, specified as a handle or a character vector representing the model name.

Example: `gcs`

**buildSpec** — Specifies the code generation action for the build process

*`'StandaloneCoderTarget'` (default) | `'ModelReferenceSimTarget'` | `'ModelReferenceCoderTarget'` | `'ModelReferenceCoderTargetOnly'` | `'CleanTopModel'`*

`buildSpec` directs the code generator to perform the selected build action for the model and build process:

- Honors the setting of the **Rebuild** parameter on the **Model Referencing** pane of the Configuration Parameters dialog box.
- Requires a Simulink Coder license only if you build a model reference Simulink Coder target, not if you build a model reference simulation target only.

The `buildSpec` argument must be one of the following.

<table>
<thead>
<tr>
<th><code>buildSpec</code></th>
<th>Build Action</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>'StandaloneCoderTarget'</code></td>
<td>Builds a standalone Simulink Coder binary executable file from <code>model</code>, using the current model configuration settings. If <code>model</code> has not been loaded, <code>slbuild</code> loads it before initiating the build process.</td>
</tr>
<tr>
<td>buildSpec</td>
<td>Build Action</td>
</tr>
<tr>
<td>---------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>'ModelReferenceSimTarget'</td>
<td>Builds a model reference simulation target (does not require a Simulink Coder license).</td>
</tr>
<tr>
<td>'ModelReferenceCoderTarget'</td>
<td>Builds a model reference Simulink Coder target and the corresponding model reference simulation target.</td>
</tr>
<tr>
<td>'ModelReferenceCoderTargetOnly'</td>
<td>Builds only a model reference Simulink Coder target.</td>
</tr>
<tr>
<td>'CleanTopModel'</td>
<td>Cleans the model build area enough to trigger regeneration of the top model code at the next build.</td>
</tr>
</tbody>
</table>

Example: 'ModelReferenceSimTarget'

**Name-Value Pair Arguments**

Specify optional comma-separated pairs of Name, Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside quotes. You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

Example: 'UpdateThisModelReferenceTarget','Force'

**UpdateThisModelReferenceTarget — Specifies a conditional rebuild option for the model reference build**

'Force' | 'IfOutOfDateOrStructuralChange' | 'IfOutOfDate'

The 'UpdateThisModelReferenceTarget' value specifies a conditional rebuild option for the model reference target build. This option applies only to model, not to any models referenced by model.

<table>
<thead>
<tr>
<th>'UpdateThisModelReferenceTarget' Value</th>
<th>Equivalent Rebuild Option</th>
<th>Rebuild Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>'Force'</td>
<td>Always</td>
<td>Unconditionally rebuilds the model.</td>
</tr>
<tr>
<td>'IfOutOfDateOrStructuralChange'</td>
<td>If any changes detected</td>
<td>Rebuilds the model if the build process detects any changes.</td>
</tr>
<tr>
<td>'IfOutOfDate'</td>
<td>If any changes in known dependencies detected</td>
<td>Rebuilds the model if the build process detects any changes in known dependencies of this model.</td>
</tr>
</tbody>
</table>
For more information on the different rebuild options, see **Rebuild**.

Example: 'UpdateThisModelReferenceTarget','Force'

**Dependencies**

To use this option, set buildSpec to 'ModelReferenceSimTarget', 'ModelReferenceCoderTarget', or 'ModelReferenceCoderTargetOnly'.

**ForceTopModelBuild — Specifies whether to force the top model of the model hierarchy to build**

false (default) | true

Setting the 'ForceTopModelBuild' value to true directs the code generator to generate code and build an executable image for the top model of the model hierarchy, regardless of model checksums and parameter settings.

Example: 'ForceTopModelBuild',true

**Dependencies**

To use this option, set buildSpec to 'StandaloneCoderTarget'.

**OpenBuildStatusAutomatically — Display build information**

false (default) | true

To display build information in the **Build Process** window, specify as true. The default is false. For more information about using the status window, see “View Build Process Status” (Simulink Coder).

The **Build Process Status** window supports parallel builds of referenced model hierarchies. Do not use the **Build Process Status** window for sequential (nonparallel) builds.

**Dependencies**

To use this option, set buildSpec to 'StandaloneCoderTarget'.
Compatibility Considerations

'StrandaloneRTWTarget', 'ModelReferenceRTWTarget', and 'ModelReferenceRTWTargetOnly' arguments are not recommended
Not recommended starting in R2019a

In R2019a, some arguments used with the slbuild function were updated. Legacy arguments are supported for backward compatibility.

To take advantage of the updated arguments:

• Use 'StandaloneCoderTarget' instead of 'StandaloneRTWTarget'
• Use 'ModelReferenceCoderTarget' instead of 'ModelReferenceRTWTarget'
• Use 'ModelReferenceCoderTargetOnly' instead of 'ModelReferenceRTWTargetOnly'

See Also
rtwbuild | rtwrebuild

Topics
“Model Reference Simulation Targets”
“What Is Acceleration?”
“Perform Acceleration”
“Share Build Artifacts for Faster Simulation and Code Generation”

Introduced before R2006a
slCharacterEncoding

Change MATLAB character set encoding

Syntax

slCharacterEncoding(encoding)
current = slCharacterEncoding()

Description

slCharacterEncoding(encoding) changes the MATLAB character set encoding to the one you specify. Use this syntax to change the current MATLAB character set encoding to be compatible with the character encoding of a model that you want to open. Close all open models or libraries before changing the MATLAB character set encoding except when changing from US-ASCII to another encoding.

Use this syntax to specify these encodings:

- 'US-ASCII'
- 'Windows-1252'
- 'ISO-8859-1'
- 'Shift_JIS'
- 'UTF-8'

If you want to use a different character encoding, start MATLAB with the appropriate locale settings for your operating system. Consult your operating system manual to change the locale setting. Simulink can support any character encoding that uses single-byte or double-byte characters.

current = slCharacterEncoding() returns the current MATLAB character set encoding.
Examples

Change Encoding for Simulink Session

Suppose that you create a model in a MATLAB session configured for 'Shift_JIS' and open it in a session configured for 'US-ASCII'. The warning message shows the encoding of the current session and the encoding used to create the model. If you encounter any problems with corrupted characters, for example when using MATLAB files associated with the model, then use the slCharacterEncoding function to change the character encoding.

Close all open models.

Find out the current encoding.

slCharacterEncoding()

ans =

'windows-1252'

Use slCharacterEncoding to change the character encoding of the current MATLAB session to match the model character encoding, for example to ASCII.

slCharacterEncoding('US-ASCII')

Reopen the model.

Input Arguments

coding — Character encoding to set
char | 'US-ASCII' | 'Windows-1252' | 'ISO-8859-1' | 'Shift_JIS' | 'UTF-8'

Character encoding to set, specified as one of these values:

- 'US-ASCII'
- 'Windows-1252'
- 'ISO-8859-1'
• 'Shift_JIS'
• 'UTF-8'

See Also

Topics
“Open a Model with Different Character Encoding”
“Save Models with Different Character Encodings”

Introduced before R2006a
sldebug
Start simulation in debug mode

Syntax
sldebug('sys')

Description
sldebug('sys') starts a simulation in debug mode. See “Debugger Command-Line Interface” for information about using the debugger.

Examples
The following command:
sldebug('vdp')
loads the Simulink example model vdp into memory and starts the simulation in debug mode. Alternatively, you can achieve the same result by using the sim command:
sim('vdp', 'debug', 'on')

See Also
sim

Introduced in R2006a
sldiagnostics

Display diagnostic information about Simulink system

Syntax

sldiagnostics('sys')
[txtRpt, sRpt] = sldiagnostics('sys')
[txtRpt, sRpt] = sldiagnostics('sys', options)
[txtRpt, sRpt] = sldiagnostics('sys', 'CompileStats')
[txtRpt, sRpt] = sldiagnostics('sys', 'RTWBuildStats')

Description

sldiagnostics('sys') displays the following diagnostic information associated with the model or subsystem specified by sys:

- Number of each type of block
- Number of each type of Stateflow object
- Number of states, outputs, inputs, and sample times of the root model.
- Names of libraries referenced and instances of the referenced blocks
- Time and additional memory used for each compilation phase of the root model

If the model specified by sys is not loaded, then sldiagnostics loads the model before performing the analysis.

The command sldiagnostics('sys', options) displays only the diagnostic information associated with the specific operations listed as options character vectors. The table below summarizes the options available and their corresponding valid input and output.

With sldiagnostics, you can input the name of a model or the path to a subsystem. For some analysis options, sldiagnostics can analyze only a root model. If you provide an incompatible input for one of these analyses, then sldiagnostics issues a warning. Finally, if you input a Simulink Library, then sldiagnostics cannot perform options that...
require a model compilation (Update Diagram). Instead, sldiagnostics issues a warning.

During the analysis, sldiagnostics will follow library links but will not follow or analyze Model References. See find_mdlrefs for more information on finding all Model blocks and referenced models in a specified model.

<table>
<thead>
<tr>
<th>Option</th>
<th>Valid Inputs</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>CountBlocks</td>
<td>root model, library, or subsystem</td>
<td>Lists all unique blocks in the system and the number of occurrences of each. This includes blocks that are nested in masked subsystems or hidden blocks.</td>
</tr>
<tr>
<td>CountSF</td>
<td>root model, library, or subsystem</td>
<td>Lists all unique Stateflow objects in the system and the number of occurrences of each.</td>
</tr>
<tr>
<td>Sizes</td>
<td>root model</td>
<td>Lists the number of states, outputs, inputs, and sample times, as well as a flag indicating direct feedthrough, used in the root model.</td>
</tr>
<tr>
<td>Libs</td>
<td>root model, library, or subsystem</td>
<td>Lists all unique libraries referenced in the root model, as well as the names and numbers of the library blocks.</td>
</tr>
<tr>
<td>CompileStats</td>
<td>root model</td>
<td>Lists the time and additional memory used for each compilation phase of the root model. This information helps users troubleshoot model compilation speed and memory issues.</td>
</tr>
<tr>
<td>RTWBuildStats</td>
<td>root model</td>
<td>Lists the same information as the CompileStats diagnostic. When issued with the second output argument sRpt, it captures the same statistics included in CompileStats and also the Simulink Coder build statistics. You must explicitly specify this option, because it is not part of the default analysis.</td>
</tr>
<tr>
<td>Option</td>
<td>Valid Inputs</td>
<td>Output</td>
</tr>
<tr>
<td>----------</td>
<td>--------------</td>
<td>--------------------------------------</td>
</tr>
<tr>
<td>All</td>
<td>not applicable</td>
<td>Performs all diagnostics.</td>
</tr>
</tbody>
</table>

**Note** Running the `CompileStats` diagnostic before simulating a model for the first time will show greater memory usage. However, subsequent runs of the `CompileStats` diagnostic on the model will require less memory usage.

```matlab
[txtRpt, sRpt] = sldiagnostics('sys')
```
returns the diagnostic information as a textual report `txtRpt` and a structure array `sRpt`, which contains the following fields that correspond to the diagnostic options:

- blocks
- stateflow
- sizes
- links
- compilestats

```matlab
[txtRpt, sRpt] = sldiagnostics('sys', options)
```
returns only the specified options. If your chosen options specify just one type of analysis, then `sRpt` contains the results of only that analysis.

```matlab
[txtRpt, sRpt] = sldiagnostics('sys', 'CompileStats')
```
returns information on time and memory usage in `txtRpt` and `sRpt`.

```matlab
[txtRpt, sRpt] = sldiagnostics('sys', 'RTWBuildStats')
```
includes Simulink Coder build statistics in addition to the information reported for `CompileStats` in the `sRpt` output.

- `txtRpt` contains the formatted textual output of time spent in each of the phases in Simulink and Simulink Coder (if you specified `RTWBuildStats`), for example:

  ```plaintext
  Compile Statistics For: rtwdemo_counter
  Cstat1: 0.00 seconds Model compilation pre-start
  Cstat2: 0.00 seconds Stateflow compile pre-start notification
  Cstat3: 0.10 seconds Post pre-comp-start engine event
  Cstat4: 10.00 seconds Stateflow compile start notification
  Cstat5: 0.00 seconds Model compilation startup completed
  ```

- `sRpt` is a MATLAB structure containing time and memory usage for each of the phases, for example:
sRpt =
Model:    'myModel1'
Statistics:    [1x134 struct]

The size of the sRpt.Statistics array indicates the number of compile and build phases executed during the operation. Examine the Statistics fields:

sRpt.Statistics(1) =
Description:    'Phase1'
CPUTime:    7.2490
WallClockTime    4.0092
ProcessMemUsage:    26.2148
ProcessMemUsagePeak:    28.6680
ProcessVMSize:    15.9531

CPUTime and WallClockTime show the elapsed time for the phase in seconds. ProcessMemUsage, ProcessMemUsagePeak and ProcessVMSize show the memory consumption during execution of the phase in MB.

Examine these key metrics to understand the performance:

• WallClockTime—The real-time elapsed in each phase in seconds. Sum the WallClockTime in each phase to get the total time taken to perform the operation:

  ElapsedTime = sum([statRpt.Statistics(:).WallClockTime]);

• ProcessMemUsage—The amount of memory consumed in each phase. Sum the ProcessMemUsage across all the phases to get the memory consumption during the entire operation:

  TotalMemory = sum([statRpt.Statistics(:).ProcessMemUsage]);

• ProcessMemUsagePeak—The maximum amount of allocated memory in each phase. Get the maximum of this metric across all the phases to find the peak memory allocation during the operation:

  PeakMemory = max([statRpt.Statistics(:).ProcessMemUsagePeak]);

**Note** Memory statistics are available only on the Microsoft Windows platform.
Examples

The following command counts and lists each type of block used in the sldemo_bounce model that comes with Simulink software.

sldiagnostics('sldemo_bounce', 'CountBlocks')

The following command counts and lists both the unique blocks and Stateflow objects used in the sf_boiler model that comes with Stateflow software; the textual report returned is captured as myReport.

myReport = sldiagnostics('sf_boiler', 'CountBlocks', 'CountSF')

The following commands open the f14 model that comes with Simulink software, and counts the number of blocks used in the Controller subsystem.

sldiagnostics('f14/Controller', 'CountBlocks')

The following command runs the Sizes and CompileStats diagnostics on the f14 model, capturing the results as both a textual report and structure array.

[txtRpt, sRpt] = sldiagnostics('f14', 'Sizes', 'CompileStats')

See Also
find_system | get_param

Introduced in R2006a
sldiagviewer.diary

Log simulation warnings and errors and build information to file

Syntax

sldiagviewer.diary
sldiagviewer.diary(filename)
sldiagviewer.diary(toggle)
sldiagviewer.diary(filename,'UTF-8')

Description

sldiagviewer.diary intercepts build information, warnings, and errors transmitted to the Command Window or the Diagnostic Viewer and logs them to a text file diary.txt in the current folder.

sldiagviewer.diary(filename) toggles the logging state of the text file specified by filename.

sldiagviewer.diary(toggle) turns logging to the log file on or off. The setting applies to the last file name you specified for logging or to diary.txt if you did not specify a file name.

sldiagviewer.diary(filename,'UTF-8') specifies the character encoding for the log file filename.

Examples

Log Build Information and Simulation Warnings and Errors

Start logging build information and simulation warnings and errors to diary.txt.
Open System
rtwbuild('vdp')
Open diary.txt to view logs.

### Starting build procedure for model: vdp
### Build procedure for model: 'vdp' aborted due to an error.
...

**Log to Specific File**

Set up logging to a file.

sldiagviewer.diary('C:\MyLogs\log1.txt')

**Toggle File Logging State**

Switch the logging state of a file.

sldiagviewer.diary('C:\MyLogs\log1.txt') % Start logging
open_system('vdp')
rtwbuild('vdp')

sldiagviewer.diary('off') % Switch off logging
open_system('sldemo_fuelsys')
rtwbuild('sldemo_fuelsys')

sldiagviewer.diary('on') % Resume logging

**Specify Log File Name and Character Encoding**

Set the file name to log to and the character encoding to use.
sldiagviewer.diary('C:\MyLogs\log1.txt','UTF-8')

**Input Arguments**

**toggle — Logging state**

'off' | 'on'

Logging state, specified as 'on' or 'off'.

Example: `sldiagviewer.diary('on')`

**filename — Name of file to log data to**

character vector

Name of file to log data to, specified as a character vector.

Example: `sldiagviewer.diary('C:\Simulations\mySimulationDiary.txt')`

**See Also**

**Topics**

“View Diagnostics”

“Customize Diagnostic Messages”

**Introduced in R2014a**
sldiscmdl

Discretize model that contains continuous blocks

Syntax

\[
\text{sldiscmdl('model\_name','sample\_time')} \\
\text{sldiscmdl('model\_name','sample\_time','method')} \\
\text{sldiscmdl('model\_name','sample\_time','options')} \\
\text{sldiscmdl('model\_name','sample\_time','method','freq')} \\
\text{sldiscmdl('model\_name','sample\_time','method','options')} \\
\text{[old\_blks,new\_blks] = sldiscmdl('model\_name','sample\_time','method','freq','options')} \\
\]

Description

\text{sldiscmdl('model\_name','sample\_time')} discretizes the model named 'model\_name' using the specified sample\_time. The model does not need to be open, and the units for sample\_time are simulation seconds.

\text{sldiscmdl('model\_name','sample\_time','method')} discretizes the model using sample\_time and the transform method specified by method.

\text{sldiscmdl('model\_name','sample\_time','options')} discretizes the model using sample\_time and criteria specified by the options cell array. This array consists of four elements: \{target, replace\_with, put\_into, prompt\}.

\text{sldiscmdl('model\_name','sample\_time','method','freq')} discretizes the model using sample\_time, method, and the critical frequency specified by freq. The units for freq are Hz. When you specify freq, method must be 'prewarp'.

\text{sldiscmdl('model\_name','sample\_time','method','options')} discretizes the model using sample\_time, method, and options.
sldiscmdl('model_name', sample_time, method, freq, options) discretizes the model using sample_time, method, freq, and options. When you specify freq, method must be 'prewarp'.

[old_blks, new_blks] = sldiscmdl('model_name', sample_time, method, freq, options) discretizes the model using sample_time, method, freq, and options. When you specify freq, method must be 'prewarp'. The function also returns two cell arrays that contain full path names of the original, continuous blocks and the new, discretized blocks.

Input Arguments

**model_name**

Name of the model to discretize.

**sample_time**

Sample-time specification for the model:

- Scalar value: Sample time with zero offset, such as 1
- Two-element vector: Sample time with nonzero offset, such as [1 0.1]

**method**

Method of converting blocks from continuous to discrete mode:

- 'zoh' (default): Zero-order hold on the inputs
- 'foh': First-order hold on the inputs
- 'tustin': Bilinear (Tustin) approximation
- 'prewarp': Tustin approximation with frequency prewarping
- 'matched': Matched pole-zero method
  - For single-input, single-output (SISO) systems only
freq

Critical frequency in Hz. This input applies only when the method input is 'prewarp'.

options

Cell array \{target, replace_with, put_into, prompt\}, where each element can take the following values:

<table>
<thead>
<tr>
<th>target</th>
<th>'all' (default)</th>
<th>Discretize all continuous blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>'selected'</td>
<td>Discretize only selected blocks in the model</td>
</tr>
<tr>
<td></td>
<td>'full_blk_path'</td>
<td>Discretize specified block</td>
</tr>
</tbody>
</table>

| replace_with | 'parammask' (default) | Create discrete blocks whose parameters derive from the corresponding continuous blocks |
|             | 'hardcoded'         | Create discrete blocks with hard-coded parameters placed directly into each block dialog box |

| put_into     | 'copy' (default)  | Create discretization in a copy of the original model |
|             | 'configurable'    | Create discretization candidate in a configurable subsystem |
|             | 'current'         | Apply discretization to the current model |
|             | 'untitled'        | Create discretization in a new untitled window |

| prompt       | 'on' (default)   | Show discretization information at the command prompt |
|             | 'off'            | Do not show discretization information at the command prompt |

Examples

Discretize all continuous blocks in the slexAircraftExample model using a 1-second sample time:
Discretize the Aircraft Dynamics Model subsystem in the slexAircraftExample model using a 1-second sample time, a 0.1-second offset, and a first-order hold transform method:

```
sldiscmdl('slexAircraftExample',1);
```

Discretize the Aircraft Dynamics Model subsystem in the slexAircraftExample model and retrieve the full path name of the second discretized block:

```
[old_blks,new_blks] = sldiscmdl('slexAircraftExample',[1 0.1],'foh',
{'slexAircraftExample/Aircraft Dynamics Model','parammask','copy','on'});
```

See Also

slmdldiscui

Topics

“Discretize a Model with the sldiscmdl Function”

Introduced before R2006a
slexpr

Generate expression to use in value of parameter object

Syntax

expressionOut = slexpr(expressionIn)

Description

expressionOut = slexpr(expressionIn) converts the MATLAB-syntax expression expressionIn to an object, expressionOut, that you can use to set the Value property of a parameter object (such as Simulink.Parameter). When you use multiple parameter objects to set block parameter values, you can use the expression to model mathematical relationships between the objects. For more information, see “Set Variable Value by Using a Mathematical Expression”.

Examples

Model Relationship Between Mass, Length, and Moment of Inertia of Metronome

In the base workspace, create three Simulink.Parameter objects that represent the mass, length, and moment of inertia of a pointlike metronome.

m = Simulink.Parameter;
r = Simulink.Parameter;
J = Simulink.Parameter;

Set the mass to 0.1 kg and the length to 1.0 m.

m.Value = 0.1;
r.Value = 1.0;

Set the value of the moment of inertia to the mass times the square of the length.

J.Value = slexpr('m*r^2');
Simulink preserves the expression, \( m*r^2 \). If you change the value of the mass or the length, Simulink recalculates the value of the moment of inertia.

**Input Arguments**

expressionIn — Target expression
string | character vector

Target expression, specified as a string or character vector.

Example: "myParam + myOtherParam"

Data Types: char | string

**Output Arguments**

expressionOut — Simulink representation of expression
Simulink.data.Expression object

Simulink representation of the target expression, returned as a Simulink.data.Expression object. A Simulink.data.Expression object has no use outside the Value property of a parameter object.

**See Also**

Simulink.Parameter

**Topics**

“Share and Reuse Block Parameter Values by Creating Variables”
“Create Tunable Calibration Parameter in the Generated Code” (Simulink Coder)

**Introduced in R2018a**
sllsFileChangedOnDisk

Determine whether model has changed since it was loaded

Syntax

```
Changed = sllsFileChangedOnDisk('sys')
```

Description

```
Changed = sllsFileChangedOnDisk('sys') Returns true if the file which contains block diagram sys was changed on disk since the block diagram was loaded.
```

Examples

```
To ensure that code is not generated for a model whose file has changed on disk since it was loaded, include the following in the 'entry' section of the STF_make_rtw_hook.m file:

if (sllsFileChangedOnDisk(sys))
    error('File has changed on disk since it was loaded. Aborting code generation.');
end
```

See Also

Topics

“Customize Build Process with STF_make_rtw_hook File” (Simulink Coder)
“Model File Change Notification”

Introduced in R2007b
slLibraryBrowser

Open Simulink Library Browser

Syntax

slLibraryBrowser
slLibraryBrowser('open')
slLibraryBrowser('noshow')
libraryhandle = slLibraryBrowser
slLibraryBrowser('close')

Description

slLibraryBrowser opens the Simulink Library Browser.

If you want to load the Simulink block library, use `load_system simulink` instead.

If you want to start Simulink without opening any windows, use the faster `start_simulink` instead.

`slLibraryBrowser('open')` opens the Library Browser.

`slLibraryBrowser('noshow')` loads the Library Browser in memory without making it visible. Use this to make future calls to `slLibraryBrowser('open')` faster.

`libraryhandle = slLibraryBrowser` returns the handle of the Library Browser object.

`slLibraryBrowser('close')` closes the Library Browser.

Examples
Open and Close the Library Browser

```matlab
slLibraryBrowser
slLibraryBrowser('close')
```

Load the Library Browser and Get a Handle

```matlab
libraryhandle = slLibraryBrowser('noshow')
```

See Also

simulink | start_simulink

Topics

“Build and Edit a Model in the Simulink Editor”
“Model Editing Environment”

Introduced in R2016a
slmdldiscui

Open Model Discretizer GUI

Syntax

slmdldiscui
slmdldiscui('model')

Description

slmdldiscui opens the Model Discretizer. A model does not need to be open.

slmdldiscui('model') opens the Model Discretizer for the model or library called 'name'.

To use the Model Discretizer, you must have a Control System Toolbox license, version 5.2 or later.

Examples

Open the Model Discretizer for the slexAircraftExample model:
slmdldiscui('slexAircraftExample')

Open the Model Discretizer for the discretizing library:
slmdldiscui('discretizing')

See Also
sldiscmdl

Topics
“Discretize a Model with the Model Discretizer”
Introduced before R2006a
**slprofreport**

Regenerate profiler report from data, ProfileData, saved from previous run

**Syntax**

```matlab
slprofreport(model_nameProfileData)
```

**Description**

When you run a model with the profiler enabled, the simulation generates the data and saves it in the variable, `model_nameProfileData`. `slprofreport(model_nameProfileData)` generates a profiler report based on the data in `model_nameProfileData`, saved from the model run.

**Input Arguments**

**ProfileData**

Variable that contains profiler data from a model run. The variable name consists of the model name and `ProfileData`, for example, `vdpProfileData`.

Default: None

**Examples**

**Regenerate Simulink Profiler Results**

Regenerate the Profiler report for model `vdp`

In the MATLAB Command Window, start the `vdp` model.

In the Simulink editor window, run `vdp` model with Simulink Profiler enabled.
Simulink stores the data to the variable `vdpProfileData`.

To review the report, in the MATLAB Command Window

```
slprofreport(vdpProfileData)
```

The Simulink Profiler Report window is displayed.

**See Also**

**Topics**

“Save Profiler Results”
“How Profiler Captures Performance Data”

**Introduced in R2012a**
slproject.create

(Not recommended) Create blank project

Note slproject.create is not recommended. Use matlab.project.createProject instead. For more information, see “Compatibility Considerations”.

Syntax

proj = slproject.create
proj = slproject.create(path)
proj = slproject.create(name)

Description

proj = slproject.create creates and opens a project using the blank project template from the Simulink start page, and returns a project object. Use the project object to manipulate the currently open project at the command line. The new project is created in the default project folder. To change the default folder for new projects, on the MATLAB Home tab, click Preferences. In the Preferences dialog box, on the MATLAB Project pane, set the Default folder.

proj = slproject.create(path) creates the project at the location specified by path.

proj = slproject.create(name) creates the project in the default folder, with the name specified by name.

Examples

Create a Blank Project in the Default Folder

slproject.create
You can control the default folder for new projects using the project preferences.

Create a Blank Project in a Specified Folder

```matlab
proj = slproject.create('C:\work\myprojectname');
```

Create a Named Blank Project in the Default Folder

```matlab
proj = slproject.create('myprojectname');
```

### Input Arguments

- **path** — Path for the new project location
  - character vector
  - Path for the new project location, specified as a character vector. If you do not specify the path, `slproject.create` creates the project in the default location. You can change the default location in the project preferences.
  - Example: `C:\work\projectname`
  - Data Types: `char`

- **name** — Name for the new project
  - character vector
  - Name for the new project, specified as a character vector.
  - Example: `myproject`
  - Data Types: `char`

### Output Arguments

- **proj** — Project
  - project object
Project, returned as a project object. Use the project object to manipulate the currently open project at the command line.

Properties of \texttt{proj} output argument.

<table>
<thead>
<tr>
<th><strong>Project Property</strong></th>
<th><strong>Description</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Project name</td>
</tr>
<tr>
<td>Information</td>
<td>Information about the project such as the description, source control integration, repository location, and whether it is a top-level project.</td>
</tr>
<tr>
<td>Dependencies</td>
<td>Dependencies between project files in a MATLAB digraph object.</td>
</tr>
<tr>
<td>Shortcuts</td>
<td>Shortcut files in the project.</td>
</tr>
<tr>
<td>ProjectPath</td>
<td>Folders that the project puts on the MATLAB path.</td>
</tr>
<tr>
<td>ProjectReferences</td>
<td>Folders that contain referenced projects. Contains read-only project objects for referenced projects.</td>
</tr>
<tr>
<td>Categories</td>
<td>Categories of project labels.</td>
</tr>
<tr>
<td>Files</td>
<td>Paths and names of project files.</td>
</tr>
<tr>
<td>RootFolder</td>
<td>Full path to project root folder.</td>
</tr>
</tbody>
</table>

**Compatibility Considerations**

**Simulink project API is not recommended**

*Not recommended starting in R2019a*

Starting in R2019a, instead of \texttt{simulinkproject} and related functions, use the \texttt{currentProject} or \texttt{openProject} functions in MATLAB. The Simulink project API will continue to be supported, but, after R2019a, new features might be available only if you use the new MATLAB project API. There are no plans to remove the Simulink project API at this time.
The new MATLAB project API is part of the R2019a functionality enabling you to use projects in MATLAB, with or without Simulink. You can now share projects with users who do not have Simulink.

See Also
Simulink.createFromTemplate | addFile | addFolderIncludingChildFiles | addPath | addReference | addShortcut | matlab.project.createProject | simulinkproject

Topics
“Creating Projects Programmatically” on page 15-685
“Automate Simulink Project Tasks Using Scripts”

Introduced in R2017a
addPath

(Not recommended) Add folder to path of project

Note simulinkproject is not recommended. Use currentProject or openProject instead. For more information, see “Compatibility Considerations”.

Syntax

folderpath = addPath(project, folder)

Description

folderpath = addPath(project, folder) adds a folder in a project to the current project path. The folder must be in the project. The project puts the folders on the MATLAB search path when it loads and removes them from the path when it closes. To learn more, see “Specify Project Path”.

Examples

Add a Folder to the Project Path

sldemo_slproject_airframe;
project = simulinkproject;

Create a new folder:

folderpath = fullfile(project.RootFolder,'folder');
mkdir(filepath);

Add this new folder to the project.

projectFile = addFile(project,folderpath);

Add this new folder to the project path.
folderpath = addPath(project, folderpath);

**Input Arguments**

**project — Project**

project object

Project, specified as a project object. Use simulinkproject to create a project object to manipulate a project at the command line.

**folder — Path of folder**

character vector | string

Path of the folder to add relative to the project root folder, specified as a character vector or string. The folder must be within the root folder.

Example: models/myfolder

**Output Arguments**

**folderpath — Path folder**

path folder object

Path folder object containing the specified folder path. The project puts the folders on the MATLAB search path when it loads and removes them from the path when it closes.

**Compatibility Considerations**

**Simulink project API is not recommended**

*Not recommended starting in R2019a*

Starting in R2019a, instead of simulinkproject and related functions, use the currentProject or openProject functions in MATLAB. The Simulink project API will continue to be supported, but, after R2019a, new features might be available only if you use the new MATLAB project API. There are no plans to remove the Simulink project API at this time.
The new MATLAB project API is part of the R2019a functionality enabling you to use projects in MATLAB, with or without Simulink. You can now share projects with users who do not have Simulink.

Compatibility Considerations

Simulink project API is not recommended
_Not recommended starting in R2019a_

Starting in R2019a, instead of `simulinkproject` and related functions, use the `currentProject` or `openProject` functions in MATLAB. The Simulink project API will continue to be supported, but, after R2019a, new features might be available only if you use the new MATLAB project API. There are no plans to remove the Simulink project API at this time.

The new MATLAB project API is part of the R2019a functionality enabling you to use projects in MATLAB, with or without Simulink. You can now share projects with users who do not have Simulink.

See Also

`addFile` | `addFolderIncludingChildFiles` | `removePath` | `simulinkproject`

Topics

“Specify Project Path”

Introduced in R2017a
removePath

(Not recommended) Remove folder from project path

**Note**  *simulinkproject* is not recommended. Use *currentProject* or *openProject* instead. For more information, see “Compatibility Considerations”.

**Syntax**

```matlab
removePath(project, folder)
```

**Description**

`removePath(project, folder)` removes a folder in a project from the current project path. The folder must be in the project.

**Examples**

**Remove a Folder from the Project Path**

```matlab
sldemo_slproject_airframe;
project = simulinkproject;
```

Create a new folder.

```matlab
folderpath = fullfile(project.RootFolder,'folder');
mkdir(filepath);
```

Add this new folder to the project.

```matlab
projectFile = addFile(project,folderpath);
```

Add the new folder to the project path.

```matlab
folderpath = addPath(project,folderpath);
```
Remove the new folder from the project path.

removePath(project,folderpath)

**Input Arguments**

*project* — Project

project object

Project, specified as a project object. Use `simulinkproject` to create a project object to manipulate a project at the command line.

*folder* — Path of folder

character vector

Path of the folder to remove relative to the project root folder, specified as a character vector. The folder must be within the root folder.

Example: `models/myfolder`

**Compatibility Considerations**

**Simulink project API is not recommended**

*Not recommended starting in R2019a*

Starting in R2019a, instead of `simulinkproject` and related functions, use the `currentProject` or `openProject` functions in MATLAB. The Simulink project API will continue to be supported, but, after R2019a, new features might be available only if you use the new MATLAB project API. There are no plans to remove the Simulink project API at this time.

The new MATLAB project API is part of the R2019a functionality enabling you to use projects in MATLAB, with or without Simulink. You can now share projects with users who do not have Simulink.

**See Also**

`addPath` | `simulinkproject`
Introduced in R2017a
addReference

(Not recommended) Add referenced project to project

Note simulinkproject is not recommended. Use currentProject or openProject instead. For more information, see “Compatibility Considerations”.

Syntax

projreference = addReference(project, folder)
projreference = addReference(project, folder, type)

Description

projreference = addReference(project, folder) adds a reference to the project specified by folder. The reference is added to the current project, project.

projreference = addReference(project, folder, type) specifies the type of reference to create. Specify relative or absolute reference.

Examples

Add a Referenced Project

Create a project and get a project object.

sldemo_slproject_airframe;
project = simulinkproject;

Create a new blank project.

projectToReference = slproject.create();

Reload the first project and add a reference to the new blank project.
reload(project);
addReference(project, projectToReference, 'absolute');

Find out if a project is a top-level project. 1 indicates a top-level project.

project.Information.TopLevel
ans =
   logical
    1

**Input Arguments**

*project* — *Project*
project object

Project, specified as a project object. Use `simulinkproject` to create a project object to manipulate a project at the command line.

*folder* — *Path of folder*
character vector

Path of the folder to add the reference, relative to the project root folder, specified as a character vector. The folder must be within the root folder.

Example: `models/myfolder`

*type* — *Type of reference*
relative | absolute

Type of reference, relative to the project root folder, specified as a character vector.

**Output Arguments**

*projreference* — *Project reference*
project reference object

Project reference object containing information about the referenced project.
Compatibility Considerations

Simulink project API is not recommended

Not recommended starting in R2019a

Starting in R2019a, instead of `simulinkproject` and related functions, use the `currentProject` or `openProject` functions in MATLAB. The Simulink project API will continue to be supported, but, after R2019a, new features might be available only if you use the new MATLAB project API. There are no plans to remove the Simulink project API at this time.

The new MATLAB project API is part of the R2019a functionality enabling you to use projects in MATLAB, with or without Simulink. You can now share projects with users who do not have Simulink.

See Also

`removeReference` | `simulinkproject`

Introduced in R2017a
removeReference

(Not recommended) Add folder to project path

**Note** `simulinkproject` is not recommended. Use `currentProject` or `openProject` instead. For more information, see “Compatibility Considerations”.

**Syntax**

`removeReference(project, folder)`

**Description**

`removeReference(project, folder)` removes the reference to the project `project` from the current project.

**Examples**

**Remove a Referenced Project**

Create a project and get a project object.

```matlab
sldemo_slproject_airframe;
project = simulinkproject;
```

Create a new blank project.

```matlab
projectToReference = slproject.create();
```

Reload the first project and add a reference to the new blank project.

```matlab
project.reload();
addReference(project, projectToReference, 'absolute');
```

Remove the reference to the blank project.
removeReference(project, projectToReference);

**Input Arguments**

**project** — Project

project object

Project, specified as a project object. Use `simulinkproject` to create a project object to manipulate a project at the command line.

**folder** — Path of folder

character vector

Path of the folder to the reference to be removed, relative to the project root folder, specified as a character vector. The folder must be within the root folder.

Example: `models/myfolder`

**Compatibility Considerations**

**Simulink project API is not recommended**

*Not recommended starting in R2019a*

Starting in R2019a, instead of `simulinkproject` and related functions, use the `currentProject` or `openProject` functions in MATLAB. The Simulink project API will continue to be supported, but, after R2019a, new features might be available only if you use the new MATLAB project API. There are no plans to remove the Simulink project API at this time.

The new MATLAB project API is part of the R2019a functionality enabling you to use projects in MATLAB, with or without Simulink. You can now share projects with users who do not have Simulink.

**See Also**

`addReference` | `simulinkproject`
Introduced in R2017a
addShortcut

(Not recommended) Add shortcut to project

Note simulinkproject is not recommended. Use currentProject or openProject instead. For more information, see “Compatibility Considerations”.

Syntax

shortcut = addShortcut(project, file)

Description

shortcut = addShortcut(project, file) adds a shortcut to the specified file in the project.

To set the shortcut to run at startup or shutdown, use Project. See “Automate Startup Tasks”.

Examples

Add a Shortcut

Create a project and get a project object.

sldemo_slproject_airframe;
project = simulinkproject;

Create a new file.

filepath = fullfile(project.RootFolder, 'new_model.slx')
new_system('new_model');
save_system('new_model', filepath)

Add this new model to the project.
projectFile = addFile(project,filepath)
Add a new shortcut to the new model.
shortcut = addShortcut(project,filepath);

Input Arguments

project — Project
project object

Project, specified as a project object. Use `simulinkproject` to create a project object to manipulate a project at the command line.

debug — Path of file
character vector

Path of the file to add a shortcut to, relative to the project root folder, including the file extension, specified as a character vector. The file must be within the root folder.

Example: `‘models/myModelName.slx’`

Output Arguments

shortcut — Shortcut
shortcut object

Shortcut object containing information about the shortcut.

Compatibility Considerations

Simulink project API is not recommended

Not recommended starting in R2019a

Starting in R2019a, instead of `simulinkproject` and related functions, use the `currentProject` or `openProject` functions in MATLAB. The Simulink project API will continue to be supported, but, after R2019a, new features might be available only if you
use the new MATLAB project API. There are no plans to remove the Simulink project API at this time.

The new MATLAB project API is part of the R2019a functionality enabling you to use projects in MATLAB, with or without Simulink. You can now share projects with users who do not have Simulink.

See Also
removeShortcut | simulinkproject

Topics
“Automate Startup Tasks”

Introduced in R2017a
removeShortcut

(Not recommended) Remove shortcut from project

Note simulinkproject is not recommended. Use currentProject or openProject instead. For more information, see “Compatibility Considerations”.

Syntax

removeShortcut(project, file)

Description

removeShortcut(project, file) removes the shortcut to the specified file in the project.

Examples

Remove a Shortcut

Create a project and get a project object.

sldemo_slproject_airframe;
project = simulinkproject;

Create a new file.

filepath = fullfile(project.RootFolder, 'new_model.slx')
new_system('new_model');
save_system('new_model', filepath)

Add this new model to the project.

projectFile = addFile(project, filepath)
Add a new shortcut to the new model.

shortcut = addShortcut(project, filepath);

Remove the shortcut.

removeShortcut(project, shortcut);

**Input Arguments**

**project — Project**

project object

Project, specified as a project object. Use `simulinkproject` to create a project object to manipulate a project at the command line.

**file — Path of file**

character vector

Path of the shortcut file, relative to the project root folder, including the file extension, specified as a character vector. The file must be within the root folder.

Example: ‘models/myModelName.slx’

**Compatibility Considerations**

**Simulink project API is not recommended**

Not recommended starting in R2019a

Starting in R2019a, instead of `simulinkproject` and related functions, use the `currentProject` or `openProject` functions in MATLAB. The Simulink project API will continue to be supported, but, after R2019a, new features might be available only if you use the new MATLAB project API. There are no plans to remove the Simulink project API at this time.

The new MATLAB project API is part of the R2019a functionality enabling you to use projects in MATLAB, with or without Simulink. You can now share projects with users who do not have Simulink.
See Also
addShortcut | simulinkproject

Introduced in R2017a
slproject.getCurrentProject

(To be removed) Manipulate current project at command line

**Note** slproject.getCurrentProject will be removed in a future release. Use currentProject instead.

**Syntax**

```matlab
proj = slproject.getCurrentProject
```

**Description**

`proj = slproject.getCurrentProject` gets the current project open in the Project Tool and returns a project object `proj` that you can use to manipulate the project programmatically. If no project is open, then you see an error.

**Note** slproject.getCurrentProject is replaced by slproject.getCurrentProjects.

**Examples**

**Get Airframe Example Project**

Open the Airframe project and use `slproject.getCurrentProject` to get a project object to manipulate the project at the command line.

```matlab
sldemo_slproject_airframe
proj = slproject.getCurrentProject
```

```matlab
proj =

    ProjectManager with properties:
```
Name: 'Simulink Project Airframe Example'
Categories: [1x1 slproject.Category]
Shortcuts: [1x8 slproject.Shortcut]
ProjectPath: [1x7 slproject.PathFolder]
ProjectReferences: [1x0 slproject.ProjectReference]
Files: [1x30 slproject.ProjectFile]
RootFolder: 'C:\Work\Simulink\Projects\slexamples\airframe'

Output Arguments

proj — Project
project object

Project, returned as a project object. Use the project object to manipulate the currently open project at the command line.

See Also

Functions
currentProject | simulinkproject | slproject.getCurrentProjects |
slproject.loadProject

Introduced in R2013a
slproject.getCurrentProjects

(Not recommended) List all top-level projects

**Note** slproject.getCurrentProjects is not recommended. Use matlab.project.rootProject instead. For more information, see “Compatibility Considerations”.

**Syntax**

```matlab
projects = slproject.getCurrentProjects
```

**Description**

`projects = slproject.getCurrentProjects` returns a list of all open top-level projects. Currently only one or zero top-level projects can be loaded. Returns an object array of 1 or 0 `ProjectManager` objects `projects` that you can use to manipulate the project programmatically. Use `slproject.getCurrentProjects` for project automation scripts.

If you execute `slproject.getCurrentProjects` inside a project shortcut, it returns only the project that the shortcut belongs to. If the shortcut belongs to a referenced project, it returns the referenced project.

**Examples**

**Get Airframe Example Project**

Open the Airframe project and use `slproject.getCurrentProjects` to get a project object to manipulate the project at the command line.

```matlab
sldemo_slproject_airframe
proj = slproject.getCurrentProjects
```
proj =

    ProjectManager with properties:

    Name: 'Simulink Project Airframe Example'
    Categories: [1x1 slproject.Category]
    Shortcuts: [1x8 slproject.Shortcut]
    ProjectPath: [1x7 slproject.PathFolder]
    ProjectReferences: [1x0 slproject.ProjectReference]
    Files: [1x30 slproject.ProjectFile]
    RootFolder: 'C:\Work\Simulink\Projects\airframe'

**Find Project Commands**

Open the airframe project and create a project object.

```matlab
sldemo_slproject_airframe
proj = slproject.getCurrentProject
```

proj =

    ProjectManager with properties:

    Name: 'Simulink Project Airframe Example'
    Categories: [1x1 slproject.Category]
    Shortcuts: [1x8 slproject.Shortcut]
    ProjectPath: [1x7 slproject.PathFolder]
    ProjectReferences: [1x0 slproject.ProjectReference]
    Files: [1x30 slproject.ProjectFile]
    RootFolder: 'C:\Work\Simulink\Projects\airframe'

Find out what you can do with your project.

```matlab
methods(proj)
```

Methods for class slproject.ProjectManager:

- addFile
- addFolderIncludingChildFiles
- close
- createCategory
- export
- findCategory
- findFile
- isLoaded
- listModifiedFiles
- refreshSourceControl
Examine Project Properties

After you get a project object, you can examine project properties.

Open the airframe project and create a project object.

```matlab
sldemo_slproject_airframe
proj = slproject.getCurrentProjects;
```

Examine the project files.

```matlab
files = proj.Files
```

```matlab
files =
1x30 ProjectFile array with properties:
    Path
    Labels
    Revision
    SourceControlStatus
```

Examine the labels of the eighth file.

```matlab
proj.Files(8).Labels
```

```matlab
ans =
Label with properties:
    File: 'C:\Work\airframe\data\system_model.sldd'
    Data: []
    DataType: 'none'
    Name: 'Design'
    CategoryName: 'Classification'
```

Get a particular file.

```matlab
myfile = findFile(proj, 'models/AnalogControl.slx')
```
myfile = 

    ProjectFile with properties:
    
    Path:    'C:\Temp\airframe\models\AnalogControl.slx'
    Labels:  [1x1 slproject.Label]
    Revision: '2'
    SourceControlStatus: Unmodified

Find out what you can do with the file.

methods(myfile)

Methods for class slproject.ProjectFile:

addLabel
removeLabel
findLabel

Output Arguments

projects — Projects
object array of 1 or 0 ProjectManager objects

Projects, returned as an object array of 1 or 0 ProjectManager objects. Use the project object to manipulate the currently open project at the command line.

Properties of ProjectManager objects in output argument.

<table>
<thead>
<tr>
<th>Project Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Project name</td>
</tr>
<tr>
<td>Categories</td>
<td>Categories of project labels</td>
</tr>
<tr>
<td>Shortcuts</td>
<td>Shortcut files in project</td>
</tr>
<tr>
<td>ProjectPath</td>
<td>Folders that the project puts on the MATLAB path</td>
</tr>
<tr>
<td>ProjectReferences</td>
<td>Folders that contain referenced projects</td>
</tr>
<tr>
<td>Files</td>
<td>Paths and names of project files</td>
</tr>
<tr>
<td>RootFolder</td>
<td>Full path to project root folder</td>
</tr>
</tbody>
</table>
Tips

Alternatively, you can use `simulinkproject` to get a project object, but `simulinkproject` also opens and gives focus to the Project Tool. Use `simulinkproject` to open projects and explore projects interactively. Use `slproject.getCurrentProjects` for project automation scripts.

Compatibility Considerations

Simulink project API is not recommended
Not recommended starting in R2019a

Starting in R2019a, instead of `simulinkproject` and related functions, use the `currentProject` or `openProject` functions in MATLAB. The Simulink project API will continue to be supported, but, after R2019a, new features might be available only if you use the new MATLAB project API. There are no plans to remove the Simulink project API at this time.

The new MATLAB project API is part of the R2019a functionality enabling you to use projects in MATLAB, with or without Simulink. You can now share projects with users who do not have Simulink.

See Also

Functions
`matlab.project.rootProject` | `simulinkproject` |
`slproject.getCurrentProject` | `slproject.loadProject`

Introduced in R2016a
slproject.loadProject

(Not recommended) Load project

**Note** simulinkproject is not recommended. Use currentProject or openProject instead. For more information, see “Compatibility Considerations”.

**Syntax**

```
slproject.loadProject(projectPath);
proj = slproject.loadProject(projectPath)
```

**Description**

`slproject.loadProject(projectPath);` loads the project specified by the .prj file or folder `projectPath` in the Project Tool, and closes any currently open project.

`proj = slproject.loadProject(projectPath)` loads the project and returns a project object `proj` for manipulating the project. Use `slproject.loadProject` for project automation scripts.

**Examples**

**Load Project**

Load a project from a folder called 'C:/projects/project1/'. Replace this path with the location of your project.

```
proj = slproject.loadProject('C:/projects/project1/')
```
Get Airframe Example Project

Open the Airframe project and use `slproject.getCurrentProjects` to get a project object to manipulate the project at the command line.

```matlab
sldemo_slproject_airframe
proj = slproject.getCurrentProjects
```

```matlab
proj = 
   ProjectManager with properties:
     Name: 'Simulink Project Airframe Example'
     Categories: [1x1 slproject.Category]
     Shortcuts: [1x8 slproject.Shortcut]
     ProjectPath: [1x7 slproject.PathFolder]
     ProjectReferences: [1x0 slproject.ProjectReference]
     Files: [1x30 slproject.ProjectFile]
     RootFolder: 'C:\Work\Simulink\Projects\airframe'
```

Find Project Commands

Get the Airframe project.

```matlab
sldemo_slproject_airframe
proj = slproject.getCurrentProjects;
```

Find project commands.

```matlab
methods(proj)
```

Methods for class `slproject.ProjectManager`:

- `addFile`   `findCategory`
- `addFolderIncludingChildFiles`   `findFile`
- `close`   `isLoaded`
- `createCategory`   `listModifiedFiles`
- `export`   `refreshSourceControl`
- `reload`
Examine Project Properties

After you get a project object, you can examine project properties.

Get the airframe project.

```matlab
sldemo_slproject_airframe
proj = slproject.getCurrentProjects;
```

Examine the project files.

```matlab
files = proj.Files
```

files =

1x30 ProjectFile array with properties:

Path
Labels
Revision
SourceControlStatus

Examine the labels of the 13th file.

```matlab
proj.Files(13).Labels
```

ans =

Label with properties:

File: 'C:\Temp\airframe\models\AnalogControl.slx'
  Data: []
  DataType: 'none'
  Name: 'Design'
  CategoryName: 'Classification'

Get a particular file by name.

```matlab
myfile = findFile(proj, 'models/AnalogControl.slx')
```

myfile =
ProjectFile with properties:

Path: 'C:\Temp\airframe\models\AnalogControl.slx'
Labels: [1x1 slproject.Label]
Revision: '2'
SourceControlStatus: Unmodified

Find out what you can do with the file.

methods(myfile)

Methods for class slproject.ProjectFile:

addLabel
removeLabel
findLabel

Input Arguments

projectPath — Full path to project file or folder
character vector

Full path to project .prj file or the path to the project root folder, specified as a character vector.

Example: 'C:/projects/project1/myProject.prj'
Example: 'C:/projects/project1/'

Output Arguments

proj — Project
project object

Project, returned as a project object. Use the project object to manipulate and explore the project at the command line.

Properties of proj output argument.
<table>
<thead>
<tr>
<th><strong>Project Property</strong></th>
<th><strong>Description</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Project name</td>
</tr>
<tr>
<td>Categories</td>
<td>Categories of project labels</td>
</tr>
<tr>
<td>Shortcuts</td>
<td>Shortcut files in project</td>
</tr>
<tr>
<td>ProjectPath</td>
<td>Folders that the project puts on the MATLAB path</td>
</tr>
<tr>
<td>ProjectReferences</td>
<td>Folders that contain referenced projects</td>
</tr>
<tr>
<td>Files</td>
<td>Paths and names of project files</td>
</tr>
<tr>
<td>RootFolder</td>
<td>Full path to project root folder</td>
</tr>
</tbody>
</table>

**Compatibility Considerations**

**Simulink project API is not recommended**

*Not recommended starting in R2019a*

Starting in R2019a, instead of `simulinkproject` and related functions, use the `currentProject` or `openProject` functions in MATLAB. The Simulink project API will continue to be supported, but, after R2019a, new features might be available only if you use the new MATLAB project API. There are no plans to remove the Simulink project API at this time.

The new MATLAB project API is part of the R2019a functionality enabling you to use projects in MATLAB, with or without Simulink. You can now share projects with users who do not have Simulink.

**See Also**

**Functions**

`simulinkproject` | `slproject.getCurrentProjects`

**Topics**

“What Are Projects?”
Introduced in R2013a
listModifiedFiles

(Not recommended) List modified files in project

**Note** simulinkproject is not recommended. Use currentProject or openProject instead. For more information, see “Compatibility Considerations”.

**Syntax**

```matlab
modifiedfiles = listModifiedFiles(proj)
```

**Description**

`modifiedfiles = listModifiedFiles(proj)` returns the list of modified project files in the project object `proj`. `listModifiedFiles` refreshes the source control statuses in the project and then returns an array of the project files which are listed in the Modified Files view of the project.

**Examples**

**Get a List of Modified Files in the Project**

Open the airframe project and create a project object.

```matlab
sldemo_slproject_airframe;
proj = simulinkproject;
```

Modify a project model file by adding an arbitrary block.

```matlab
open_system('AnalogControl')
add_block('built-in/SubSystem', 'AnalogControl/test')
save_system('AnalogControl')
```

Get all the modified files in the project.
modifiedfiles = listModifiedFiles(proj)

modifiedfiles =

1x2 ProjectFile array with properties:

Path
Labels
Revision
SourceControlStatus

Observe two modified files. Compare with the Modified Files view in Project, where you can see a modified model file, and the corresponding .SimulinkProject definition file.

Get the second modified file.

modifiedfiles(2)

ans =

ProjectFile with properties:

Path: 'C:\Work\temp\slexamples\airframe2\models\AnalogControl.slx'
    Labels: [1x1 slproject.Label]
    Revision: '2'
    SourceControlStatus: Modified

Observe the file SourceControlStatus property is Modified. Similarly, listModifiedFiles returns any files that are added, conflicted, deleted, etc., that show up in the Modified Files view in Project.

Get all the project files with a particular source control status. For example, get the files that are Unmodified.

proj.Files(ismember([proj.Files.SourceControlStatus], matlab.sourcecontrol.Status.Unmodified))

ans =

1x29 ProjectFile array with properties:

Path
Labels
Input Arguments

proj — Project
project

Project, specified as a project object. Use `simulinkproject` to create a project object to manipulate a project at the command line.

Output Arguments

modifiedfiles — Modified files
file object | array

Modified files, returned as an array of file objects.

Compatibility Considerations

Simulink project API is not recommended

Not recommended starting in R2019a

Starting in R2019a, instead of `simulinkproject` and related functions, use the `currentProject` or `openProject` functions in MATLAB. The Simulink project API will continue to be supported, but, after R2019a, new features might be available only if you use the new MATLAB project API. There are no plans to remove the Simulink project API at this time.

The new MATLAB project API is part of the R2019a functionality enabling you to use projects in MATLAB, with or without Simulink. You can now share projects with users who do not have Simulink.
See Also

Functions
refreshSourceControl | simulinkproject

Introduced in R2016a
listRequiredFiles

(Not recommended) Get project file dependencies

Note simulinkproject is not recommended. Use currentProject or openProject instead. For more information, see “Compatibility Considerations”.

Syntax

files = listRequiredFiles(proj,file)

Description

files = listRequiredFiles(proj,file) returns the files that the specified file requires to run.

Examples

Get Required Files

Open the airframe project, create a project object and get a file.

    sldemo_slproject_airframe;
    proj = simulinkproject;
    file = 'models/slproject_f14.slx'

Get the files required by the specified file.
files = listRequiredFiles(proj, file);

### Input Arguments

**proj** — Project  
project object  

Project, specified as a project object. Use `simulinkproject` to create a project object to manipulate a project at the command line.

**file** — Path of file  
character vector | project file object

Path of the file relative to the project root folder, including the file extension, specified as a character vector, an absolute file path or an instance of a project file object. The file must be within the root folder.

Example: ‘models/myModelName.slx’

### Output Arguments

**file** — Project file  
cell array of character vectors

Required files, returned as a cell array of character vectors.

### Compatibility Considerations

**Simulink project API is not recommended**  
Not recommended starting in R2019a

Starting in R2019a, instead of `simulinkproject` and related functions, use the `currentProject` or `openProject` functions in MATLAB. The Simulink project API will continue to be supported, but, after R2019a, new features might be available only if you use the new MATLAB project API. There are no plans to remove the Simulink project API at this time.
The new MATLAB project API is part of the R2019a functionality enabling you to use projects in MATLAB, with or without Simulink. You can now share projects with users who do not have Simulink.

**See Also**

simulinkproject

**Topics**

“Perform Impact Analysis”

**Introduced in R2017a**
**refreshSourceControl**

(Not recommended) Update source control status of project files

**Note** simulinkproject is not recommended. Use currentProject or openProject instead. For more information, see “Compatibility Considerations”.

**Syntax**

refreshSourceControl(proj)

**Description**

refreshSourceControl(proj) updates the source control status for all files in the project proj. Use this to get the latest source control information before querying the SourceControlStatus property on individual files.

If you use listModifiedFiles to find all modified files in the project, you do not need to call refreshSourceControl first.

**Examples**

**Refresh Source Control Information on Files in the Project**

Open the airframe project and create a project object.

```matlab
sldemo_slproject_airframe;
proj = simulinkproject;
```

Refresh source control status before querying individual files.
refreshSourceControl(proj)

**Input Arguments**

proj — Project

Project, specified as a project object. Use `simulinkproject` to create a project object to manipulate a project at the command line.

**Compatibility Considerations**

**Simulink project API is not recommended**

*Not recommended starting in R2019a*

Starting in R2019a, instead of `simulinkproject` and related functions, use the `currentProject` or `openProject` functions in MATLAB. The Simulink project API will continue to be supported, but, after R2019a, new features might be available only if you use the new MATLAB project API. There are no plans to remove the Simulink project API at this time.

The new MATLAB project API is part of the R2019a functionality enabling you to use projects in MATLAB, with or without Simulink. You can now share projects with users who do not have Simulink.

**See Also**

**Functions**

`listModifiedFiles` | `simulinkproject`

**Introduced in R2016a**
solverprofiler.profileModel

Examine model for performance analysis

Syntax

res = solverprofiler.profileModel(model)
res = solverprofiler.profileModel(model, Name,Value)

Description

res = solverprofiler.profileModel(model) runs the Solver Profiler on the specified model and stores the results in res

res = solverprofiler.profileModel(model, Name,Value) specifies the Solver Profiler parameters using one or more Name, Value pair arguments.

Examples

Examine a Model with Default Settings

Examine the model f14 using the default commandline settings.

model = 'f14';
res = solverprofiler.profileModel(model);

You can see a summary of the results by calling res.summary.

res.summary

struct with fields:
    solver: 'ode45'
    tStart: 0
    tStop: 60
    absTol: 1.0000e-06
Open the results in the Solver Profiler dialog to visualize them. This step is equivalent to enabling OpenSP when calling the function.

solverprofiler.exploreResult(res)

**Configure Solver Profiler and Examine a Model**

Examine the model `ssc_actuator_custom_pneumatic` with a fully specified configuration.

```matlab
model = 'ssc_actuator_custom_pneumatic';
res = solverprofiler.profileModel(model, ...  
    'SaveStates', 'on', ...  
    'SaveSimscapeStates', 'On', ...  
    'SaveJacobian', 'On', ...  
    'StartTime', 5, ...  
    'StopTime', 50, ...  
    'BufferSize', 10000,...  
    'TimeOut', 5,...  
    'OpenSP', 'On',...  
    'DataFullFile', fullfile(pwd, 'ssc_profiling_result.mat'));
```

**Input Arguments**

- **model** — Model to examine
  character vector (default)

Name of model to be profiled, specified as a character vector.

Example: `h = solverprofiler.profileModel('vdp')`
Name-Value Pair Arguments

Specify optional comma-separated pairs of Name, Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside quotes. You can specify several name and value pair arguments in any order as Name1, Value1, ..., NameN, ValueN.

Example: 'StartTime', 0, 'StopTime', 10, 'SaveStates', 'On'

SaveStates — Save model states to file
off (default) | on

By default, the profiler does not save the states of the model. Enabling this parameter configures the profiler to save the states to a MAT-file.

Example: 'SaveStates', 'On'

SaveSimscapeStates — Save Simscape states to file
off (default) | on

Enable this parameter to save Simscape states to a MAT-file.

Example: 'SaveSimscapeStates', 'On'

SaveJacobian — Save model Jacobian
off (default) | on

Option to log the solver Jacobian matrices to memory. This option is useful for simulations that use implicit solvers. For a comparison of solvers, see “Compare Solvers”.

Example: 'SaveJacobian', 'On'

StartTime — Profiler start time
model start time (default) | scalar

Time, in seconds, of the simulation that the profiler starts analyzing the model. This is not the same as the start time of the simulation.

Example: 'StartTime', 5

StopTime — Profiler stop time
model stop time (default) | scalar

Time, in seconds, of the simulation to which the profiler should profile the model. By default, the analysis continues until the end of the simulation. Changing this parameter
does not change the stop time of the model which you specify in the Model Configuration Parameters.

A value less than the configured stop time of the model stops the profiling and simulation at StopTime.

Example: 'StopTime', 30

**BufferSize — Memory impact of logging**

50000 (default) | positive scalar

Maximum number of events that are logged. If the number of logged events reaches this value and memory is available, increase BufferSize. If memory is limited, consider lowering the value.

Example: 'BufferSize', 60000

**TimeOut — Maximum time to wait for solver to resume**

positive scalar

Time, in seconds, to wait before the profiler stops running. This option is useful in situations where the simulation is unable to proceed. The profiler waits for the specified time and quits if no progress has been made.

Example: 'TimeOut', 10

**OpenSP — Open the Solver Profiler dialog box**

off (default) | on

Option to open the Solver Profiler dialog box after profiling has completed.

Example: 'OpenSP', 'On'

**DataFullFile — Path and name of saved results**

character vector of full file path

By default, the profiling results are saved in a MAT-file named `model_@_dd_Month_yyyy_hh_mm_ss.mat` in the current working folder. You can specify a different file name by which to save the results in the current working folder. To save the file in a different location, specify the full path of the file, including the file name.

Example: 'DataFullFile', 'C:\Users\myusername\Documents\profiled\vdp_results.mat'
Output Arguments

**res — High-level summary of profiling results**
structure

Profiling results, returned as a structure with the fields:

**file — Full path and name of saved results**
character vector

Path and name of the MAT-file where the results of the profiling operation are stored as MAT file. By default, they are stored in the current working folder with a file name having the pattern: `model_@_dd_Month_yyyy_hh_mm_ss.mat`. To store them in a different location or by a different name, specify `DataFullFile` when calling `solverprofiler.profileModel`.

**summary — Summary of profiling results**
structure

A high-level summary of the results of the profiling operation, returned as a structure. The summary provides an overview of the performance of the simulation and health of the model.

The `summary` structure contains these fields.

<table>
<thead>
<tr>
<th>Field</th>
<th>Purpose</th>
<th>Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>solver</td>
<td>Solver used by simulation</td>
<td>any of the solvers supported by Solver Profiler</td>
<td>Solver used by the simulation as configured in the Configuration Parameters for the model. For a list of all the solvers, see “Solver”. The Solver Profiler does not support models without any continuous states.</td>
</tr>
<tr>
<td>tStart</td>
<td>Start time of simulation</td>
<td>scalar</td>
<td>Start time, in seconds, for the simulation of the model during the profiling operation.</td>
</tr>
<tr>
<td>Field</td>
<td>Purpose</td>
<td>Values</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>----------------------------------------------</td>
<td>------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>tStop</td>
<td>Stop time of simulation</td>
<td>scalar</td>
<td>Stop time, in seconds, of the simulation during the profiling operation. If StopTime is set to be earlier than the configured Stop Time of the model, the simulation stops at StopTime.</td>
</tr>
<tr>
<td>abs Tol</td>
<td>Absolute tolerance of the solver</td>
<td>positive scalar</td>
<td>Absolute tolerance of the solver as specified in the configuration settings for the model. For more information, see “Absolute tolerance”</td>
</tr>
<tr>
<td>rel Tol</td>
<td>Relative tolerance of the solver</td>
<td>positive scalar</td>
<td>Relative tolerance of the solver as specified in the configuration settings of the model. For more information, see “Relative tolerance”</td>
</tr>
<tr>
<td>hMax</td>
<td>Maximum step size</td>
<td>positive scalar</td>
<td>Largest time step that the solver can take. See “Max step size”.</td>
</tr>
<tr>
<td>hAverage</td>
<td>Average step size</td>
<td>positive scalar</td>
<td>Average size of the time step taken by the solver.</td>
</tr>
<tr>
<td>steps</td>
<td>Total steps taken</td>
<td>positive scalar</td>
<td>Total number of time steps taken by the solver.</td>
</tr>
<tr>
<td>profileTime</td>
<td>Time to profile</td>
<td>positive scalar</td>
<td>Time, in seconds, taken by the SolverProfiler to examine the model.</td>
</tr>
<tr>
<td>zcNumber</td>
<td>Total number of zero crossings</td>
<td>nonnegative scalar</td>
<td>Number of times zero crossings occur during the simulation of the model. The detection of these zero crossings incurs computational cost and can slow down the simulation. For information on zero-crossing detection, see “Zero-Crossing Detection”.</td>
</tr>
<tr>
<td>resetN</td>
<td>Number of solver resets</td>
<td>nonnegative scalar</td>
<td>Number of times the solver has to reset its parameters.</td>
</tr>
<tr>
<td>Field</td>
<td>Purpose</td>
<td>Values</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------</td>
<td>----------------------------------------------</td>
<td>-------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>jacobiNumber</td>
<td>Number of Jacobian updates</td>
<td>nonnegative scalar</td>
<td>Number of times the solver Jacobian matrix is updated during a simulation. For more information, see “Explicit Versus Implicit Continuous Solvers”.</td>
</tr>
<tr>
<td>exceptionNumber</td>
<td>Number of solver exceptions</td>
<td>nonnegative scalar</td>
<td>Total number of solver exceptions encountered during a simulation. These exceptions are events where the solver is unable to solve the model states to the specified accuracy. As a result, the solver runs adjusted trials which increase computational cost.</td>
</tr>
</tbody>
</table>

Data Types: struct

See Also
“Understand Profiling Results” | “Examine Model Dynamics Using Solver Profiler”

Topics
“Solver Selection Criteria”
“Choose a Solver”

Introduced in R2017b
start_simulink

Start Simulink without opening any windows

Syntax

start_simulink

Description

start_simulink starts Simulink without opening any models, the Start Page, or the Simulink Library Browser. Use this in startup scripts to start Simulink without any other window taking the focus away from the MATLAB Desktop. For example, use start_simulink in the MATLAB startup.m file, when starting MATLAB with the -r command line option, or in project startup scripts. Opening a model for the first time in a MATLAB session is much quicker after running start_simulink.

If you want to open the Simulink Start Page to create or open models, use the simulink function instead.

If you want to open the Library Browser, use slLibraryBrowser.

Examples

Start Simulink When Starting MATLAB

Use the -r command line option to start Simulink when starting MATLAB, without opening any windows.

On Windows, create a desktop shortcut with the following target:

```
matlabroot\bin\win64\matlab.exe -r start_simulink
```

On Linux® and Mac, enter:
matlab -r start_simulink

See Also

simulink | simulinkproject | slLibraryBrowser

Topics

“Automate Startup Tasks”

Introduced in R2015b
**slupdate**

Replace blocks from previous releases with latest versions

**Compatibility**

**Note** slupdate will be removed in a future release. The slupdate command can only upgrade some parts of your model. Use the Upgrade Advisor instead. See “Model Upgrades”.

**Syntax**

\[ \text{slupdate('sys')} \]
\[ \text{slupdate('sys', prompt)} \]
\[ \text{AnalysisResult} = \text{slupdate('sys', 'OperatingMode', 'Analyze')} \]

**Description**

\text{slupdate('sys')} \) replaces blocks in model \text{sys} \) from a previous release of Simulink software with the latest versions. The \text{slupdate} \) function alone cannot perform all upgrade checks on your model. Use the Upgrade Advisor to access the \text{slupdate} \) checks and also advice and fixes for all other upgrade checks. See “Model Upgrades”.

**Note** Best practice is to first open the model, and press CTRL+D to update the model, before you call \text{slupdate}.

\text{slupdate('sys', 'prompt')} \) specifies whether to prompt you before replacing a block. If \text{prompt} \) equals 1, the command prompts you before replacing the block. The prompt asks whether you want to replace the block. Valid responses are

- \text{y} \\
  Replace the block (the default).
Do not replace the block.

Replace this and all subsequent obsolete blocks without further prompting.

If `prompt` equals 0, the command replaces all obsolete blocks without prompting you.

In addition to replacing obsolete blocks, `slupdate`

- Reconnects broken links to masked blocks in libraries provided by MathWorks to ensure that the model reflects changes made to the blocks in this release. This will overwrite any custom changes you made to the masks of these blocks.
- Updates obsolete configuration settings for the model.

```matlab
AnalysisResult = slupdate('sys', 'OperatingMode', 'Analyze')
```
performs only the analysis portion without updating or changing the model. This command analyzes referenced models, linked libraries, and S-functions, and then returns a data structure with the following fields:

- **Message** — character vector containing a message summarizing the results
- **blockList** — cell array listing blocks that need to be updated
- **blockReasons** — cell array listing reasons for updating the corresponding blocks
- **modelList** — cell array listing referenced models and the parent model
- **libraryList** — cell array listing non-MathWorks libraries referenced
- **configSetList** — for internal use
- **sfunList** — cell array listing S-functions referenced
- **sfunOK** — logical array representing S-function status, where false indicates that an S-function needs updating and true indicates otherwise
- **sfunType** — cell array listing apparent S-function type (e.g., .mex)

**See Also**
upgradeadvisor

**Topics**
"Model Upgrades"
Introduced before R2006a
stringtype

Create string data type

Syntax

\[ \text{string} = \text{stringtype}(\text{maximum\_length}) \]
\[ \text{stringtype}(\text{maximum\_length}) \]

Description

\[ \text{string} = \text{stringtype}(\text{maximum\_length}) \] creates a Simulink string data type with a maximum length. Alternatively, you can also create string data types using the String Constant, String Concatenate, and Compose String blocks.

\[ \text{stringtype}(\text{maximum\_length}) \] creates a Simulink string data type with a maximum length that you can type directly on the MATLAB command line or in the Output data type parameter of the String Constant, String Concatenate, or Compose String block.

**Tip** Specifying \[ \text{stringtype}(0) \] creates a string data type with no maximum length. This syntax is the same as specifying string for Simulink block data types.

Examples

**Create a String Data Type of Maximum Length 10**

Create a string data type of maximum length 10.

\[ h = \text{stringtype}(10) \]

\[ h = \]

    StringType with properties:
Input Arguments

maximum_length — Maximum length
scalar

Maximum length of string data type, specified as a scalar, from 1 to 32766. This value can be an integer, MATLAB variable, or MATLAB expression.

Data Types: double

Output Arguments

string — String data type object
scalar

String object, specified as a scalar.

See Also

ASCII to String | Compose String | Scan String | String Compare | String Concatenate | String Constant | String Find | String Length | String To ASCII | String To Enum | String to Double | String to Single | Substring | To String

Topics

“Simulink Strings”

Introduced in R2018a
trim

Find trim point of dynamic system

Syntax

\[
[x,u,y,dx] = \text{trim}('sys')
\]
\[
[x,u,y,dx] = \text{trim}('sys',x0,u0,y0)
\]
\[
[x,u,y,dx] = \text{trim}('sys',x0,u0,y0,ix,iu,iy)
\]
\[
[x,u,y,dx] = \text{trim}('sys',x0,u0,y0,ix,iu,iy,dx0,idx)
\]
\[
[x,u,y,dx,options] = \text{trim}('sys',x0,u0,y0,ix,iu,iy,dx0,idx,options)
\]
\[
[x,u,y,dx,options] = \text{trim}('sys',x0,u0,y0,ix,iu,iy,dx0,idx,options,t)
\]

Description

A trim point, also known as an equilibrium point, is a point in the parameter space of a dynamic system at which the system is in a steady state. For example, a trim point of an aircraft is a setting of its controls that causes the aircraft to fly straight and level. Mathematically, a trim point is a point where the system's state derivatives equal zero. trim starts from an initial point and searches, using a sequential quadratic programming algorithm, until it finds the nearest trim point. You must supply the initial point implicitly or explicitly. If trim cannot find a trim point, it returns the point encountered in its search where the state derivatives are closest to zero in a min-max sense; that is, it returns the point that minimizes the maximum deviation from zero of the derivatives. trim can find trim points that meet specific input, output, or state conditions, and it can find points where a system is changing in a specified manner, that is, points where the system's state derivatives equal specific nonzero values.

\[
[x,u,y,dx] = \text{trim}('sys')
\]
finds the equilibrium point of the model 'sys', nearest to the system's initial state, \(x0\). Specifically, \text{trim} finds the equilibrium point that minimizes the maximum absolute value of \([x-x0,u,y]\). If \text{trim} cannot find an equilibrium point near the system's initial state, it returns the point at which the system is nearest to equilibrium. Specifically, it returns the point that minimizes \(\text{abs}(dx)\) where \(dx\) represents the derivative of the system. You can obtain \(x0\) using this command.

\[
[sizes,x0,xstr] = \text{sys}([],[],[],[],0)
\]
[x,u,y,dx] = trim('sys',x0,u0,y0) finds the trim point nearest to x0, u0, y0, that is, the point that minimizes the maximum value of
abs([x-x0; u-u0; y-y0])

[x,u,y,dx] = trim('sys',x0,u0,y0,ix,iu,iy) finds the trim point closest to x0, u0, y0 that satisfies a specified set of state, input, and/or output conditions. The integer vectors ix, iu, and iy select the values in x0, u0, and y0 that must be satisfied. If trim cannot find an equilibrium point that satisfies the specified set of conditions exactly, it returns the nearest point that satisfies the conditions, namely,
abs([x(ix)-x0(ix); u(iu)-u0(iu); y(iy)-y0(iy)])

[x,u,y,dx] = trim('sys',x0,u0,y0,ix,iu,iy,dx0,idx) finds specific nonequilibrium points, that is, points at which the system's state derivatives have some specified nonzero value. Here, dx0 specifies the state derivative values at the search's starting point and idx selects the values in dx0 that the search must satisfy exactly.

[x,u,y,dx,options] = trim('sys',x0,u0,y0,ix,iu,iy,dx0,idx,options) specifies an array of optimization parameters that trim passes to the optimization function that it uses to find trim points. The optimization function, in turn, uses this array to control the optimization process and to return information about the process. trim returns the options array at the end of the search process. By exposing the underlying optimization process in this way, trim allows you to monitor and fine-tune the search for trim points.

The following table describes how each element affects the search for a trim point. Array elements 1, 2, 3, 4, and 10 are particularly useful for finding trim points.

<table>
<thead>
<tr>
<th>No.</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>Specifies display options. 0 specifies no display; 1 specifies tabular output; -1 suppresses warning messages.</td>
</tr>
<tr>
<td>2</td>
<td>10^-4</td>
<td>Precision the computed trim point must attain to terminate the search.</td>
</tr>
<tr>
<td>3</td>
<td>10^-4</td>
<td>Precision the trim search goal function must attain to terminate the search.</td>
</tr>
<tr>
<td>4</td>
<td>10^-6</td>
<td>Precision the state derivatives must attain to terminate the search.</td>
</tr>
<tr>
<td>5</td>
<td>N/A</td>
<td>Not used.</td>
</tr>
<tr>
<td>No.</td>
<td>Default</td>
<td>Description</td>
</tr>
<tr>
<td>-----</td>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td>6</td>
<td>N/A</td>
<td>Not used.</td>
</tr>
<tr>
<td>7</td>
<td>N/A</td>
<td>Used internally.</td>
</tr>
<tr>
<td>8</td>
<td>N/A</td>
<td>Returns the value of the trim search goal function (λ in goal attainment).</td>
</tr>
<tr>
<td>9</td>
<td>N/A</td>
<td>Not used.</td>
</tr>
<tr>
<td>10</td>
<td>N/A</td>
<td>Returns the number of iterations used to find a trim point.</td>
</tr>
<tr>
<td>11</td>
<td>N/A</td>
<td>Returns the number of function gradient evaluations.</td>
</tr>
<tr>
<td>12</td>
<td>0</td>
<td>Not used.</td>
</tr>
<tr>
<td>13</td>
<td>0</td>
<td>Number of equality constraints.</td>
</tr>
<tr>
<td>14</td>
<td>100*(Number of variables)</td>
<td>Maximum number of function evaluations to use to find a trim point.</td>
</tr>
<tr>
<td>15</td>
<td>N/A</td>
<td>Not used.</td>
</tr>
<tr>
<td>16</td>
<td>$10^{-8}$</td>
<td>Used internally.</td>
</tr>
<tr>
<td>17</td>
<td>0.1</td>
<td>Used internally.</td>
</tr>
<tr>
<td>18</td>
<td>N/A</td>
<td>Returns the step length.</td>
</tr>
</tbody>
</table>

$[x,u,y,dx,options] = \text{trim('sys',x0,u0,y0,ix,iu,iy,dx0,idx,options,t)}$ sets the time to $t$ if the system is dependent on time.

**Note** If you fix any of the state, input or output values, `trim` uses the unspecified free variables to derive the solution that satisfies these constraints.

### Examples

Consider a linear state-space system modeled using a State-Space block

\[
\dot{x} = Ax + Bu \\
y = Cx + Du
\]

The $A$, $B$, $C$, and $D$ matrices to enter at the command line or in the block parameters dialog are:
Example 1

To find an equilibrium point in this model called sys, use:

\[
[x,u,y,dx,options] = \text{trim('sys')}
\]

\[
\begin{bmatrix}
0 \\
0 \\
\end{bmatrix}
\]

\[
\begin{bmatrix}
0 \\
\end{bmatrix}
\]

\[
\begin{bmatrix}
0 \\
0 \\
\end{bmatrix}
\]

\[
\begin{bmatrix}
0 \\
0 \\
\end{bmatrix}
\]

The number of iterations taken is:

\[
\text{options}(10)
\]

\[
\begin{bmatrix}
7
\end{bmatrix}
\]

Example 2

To find an equilibrium point near \( x = [1;1], \ u = [1;1] \), enter

\[
x0 = [1;1];
\]

\[
u0 = [1;1];
\]

\[
[x,u,y,dx,options] = \text{trim('sys', x0, u0)};
\]

\[
\begin{bmatrix}
1.0e-13 * \\
-0.5160 \\
-0.5169 \\
\end{bmatrix}
\]

\[
\begin{bmatrix}
0.3333 \\
0.0000 \\
\end{bmatrix}
\]
y =
-1.0000
0.3333
dx =
1.0e-12 *
0.1979
0.0035

The number of iterations taken is

options(10)
ans =
   25

Example 3

To find an equilibrium point with the outputs fixed to 1, use:

y = [1;1];
iy = [1;2];
[x,u,y,dx] = trim('sys', [], [], y, [], iy, iy)
x =
   0.0009
  -0.3075
u =
  -0.5383
   0.0004
y =
   1.0000
   1.0000
dx =
1.0e-15 *
  -0.0170
   0.1483

Example 4

To find an equilibrium point with the outputs fixed to 1 and the derivatives set to 0 and 1, use

y = [1;1];
iy = [1;2];
dx = [0;1];
idx = [1;2];
[x,u,y,dx,options] = trim('sys',[],[],y,[],[],iy,dx,idx)
x =
  0.9752  
 -0.0827
u =
 -0.3884
 -0.0124
y =
  1.0000
 1.0000
dx =
 0.0000
 1.0000

The number of iterations taken is

options(10)
an =
 13

Limitations

The trim point found by trim starting from any given initial point is only a local value. Other, more suitable trim points may exist. Thus, if you want to find the most suitable trim point for a particular application, it is important to try a number of initial guesses for x, u, and y.

Algorithms

trim uses a sequential quadratic programming algorithm to find trim points. See “Sequential Quadratic Programming (SQP)” (Optimization Toolbox) for a description of this algorithm.

Introduced before R2006a
tunablevars2parameterobjects

Create Simulink parameter objects from tunable parameters

Syntax

tunablevars2parameterobjects ('modelName')
tunablevars2parameterobjects ('modelName', class)

Description

tunablevars2parameterobjects ('modelName') creates Simulink.Parameter objects in the base workspace for the variables listed in the specified model's Tunable Parameters dialog, then deletes the source information from the dialog. To preserve the information, save the resulting Simulink parameter objects into a MAT-file.

If a tunable variable is already defined as a numeric variable in the base workspace, the variable will be replaced by a parameter object and the original variable will be copied to the object's Value property.

If a tunable variable is already defined as a Simulink parameter object, the object will not be modified but the information for the variable will still be deleted from the Tunable Parameters dialog.

If a tunable variable is defined as any other class of variable, the variable will not be modified and the information for the variable will not be deleted from the Tunable Parameters dialog.

tunablevars2parameterobjects ('modelName', class) creates objects of the specified class rather than Simulink.Parameter objects.

Input Arguments

modelName

Model name or handle
class

Parameter class to use for creating objects

Default: Simulink.Parameter

See Also

Simulink.Parameter

Topics

“Tunable Parameters”

Introduced in R2007b
ufix

Create Simulink.NumericType object describing unsigned fixed-point data type

Syntax

a = ufix(WordLength)

Description

ufix(WordLength) returns a Simulink.NumericType object that describes an unsigned fixed-point data type with the specified word length and unspecified scaling.

Note  ufix is a legacy function. In new code, use fixdt instead. In existing code, replace ufix(WordLength) with fixdt(0,WordLength).

Examples

Define a 16-bit unsigned fixed-point data type.

a = ufix(16)

a =

NumericType with properties:

  DataTypeMode: 'Fixed-point: unspecified scaling'
  Signedness: 'Unsigned'
  WordLength: 16
  IsAlias: 0
  DataScope: 'Auto'
  HeaderFile: ''
  Description: ''
See Also
Simulink.NumericType | fixdt | float | sfix | sfrac | sint | ufrac | uint

Introduced before R2006a
**ufrac**

Create `Simulink.NumericType` object describing unsigned fractional data type

**Syntax**

```matlab
a = ufrac(WordLength)
a = ufrac(WordLength, GuardBits)
```

**Description**

`ufrac(WordLength)` returns a `Simulink.NumericType` object that describes the data type of an unsigned fractional data type with a word size given by `WordLength`.

`ufrac(WordLength, GuardBits)` returns a `Simulink.NumericType` object that describes the data type of an unsigned fractional data type. The total word size is given by `WordLength` with `GuardBits` bits located to the left of the binary point.

**Note**  
`ufrac` is a legacy function. In new coder, use `fixdt` instead. In existing code, replace `ufrac(WordLength)` with `fixdt(0,WordLength,WordLength)` and `ufrac(WordLength,GuardBits)` with `fixdt(0,WordLength,(WordLength-GuardBits))`.

**Examples**

Define an 8-bit unsigned fractional data type with 4 guard bits. Note that the range of this data type is from 0 to $(1 - 2^{-8}).2^4 = 15.9375$.

```matlab
a = ufrac(8,4)
a =
```

`NumericType with properties:

```
DataTypeMode: 'Fixed-point: binary point scaling'
Signedness: 'Unsigned'
```
WordLength: 8
FractionLength: 4
IsAlias: 0
DataScope: 'Auto'
HeaderFile: ''
Description: ''

See Also
Simulink.NumericType | fixdt | float | sfix | sfrac | sint | ufix | uint

Introduced before R2006a
**uint**

Create `Simulink.NumericType` object describing unsigned integer data type

**Syntax**

```matlab
a = uint(WordLength)
```

**Description**

`uint(WordLength)` returns a `Simulink.NumericType` object that describes the data type of an unsigned integer with a word size given by `WordLength`.

**Note** `uint` is a legacy function. In new code, use `fixdt` instead. In existing code, replace `uint(WordLength)` with `fixdt(0,WordLength,0)`.

**Examples**

Define a 16-bit unsigned integer:

```matlab
a = uint(16)
```

```
NumericType with properties:
    DataTypeMode: 'Fixed-point: binary point scaling'
    Signedness: 'Unsigned'
    WordLength: 16
    FractionLength: 0
    IsAlias: 0
    DataScope: 'Auto'
    HeaderFile: ''
    Description: ''
```
See Also
Simulink.NumericType | fixdt | float | sfix | sfrac | sint | ufix | ufrac

Introduced before R2006a
unpack

Extract signal logging objects from signal logs and write them into MATLAB workspace

Syntax

log.unpack
tsarray.unpack
log.unpack('systems')
log.unpack('all')

Description

**Note** The ModelDataLogs class is supported for backwards compatibility. Starting in R2016a, you cannot log data in the ModelDataLogs format. Signal logging uses the Dataset format. In R2016a or later, when you open a model from an earlier release that had used ModelDataLogs format, the model simulated in use Dataset format.

You can convert signal logging data from ModelDataLogs to Dataset format. Converting to Dataset format makes it easier to post-process with other logged data (for example, logged states), which can also use Dataset format. For more information, see “Convert Logged Data to Dataset Format”.

If you have legacy code that uses the ModelDataLogs API, you can encounter situations that require updates to your code or model. See “Migrate Scripts That Use Legacy ModelDataLogs API”.

log.unpack or unpack(log) extracts the top level elements of the Simulink.ModelDataLogs or Simulink.SubsysDataLogs object named log (e.g., logsout).

log.unpack('systems') or unpack(log, 'systems') extracts Simulink.Timeseries and Simulink.TsArray objects from the Simulink.ModelDataLogs or Simulink.SubsysDataLogs object named log. This command does not extract Simulink.Timeseries objects from Simulink.TsArray.
objects nor does it write intermediate `Simulink.ModelDataLogs` or `Simulink.SubsysDataLogs` objects to the MATLAB workspace.

`log.unpack('all')` or `unpack(log, 'all')` extracts all the `Simulink.Timeseries` objects contained by the `Simulink.ModelDataLogs`, `Simulink.TsArray`, or `Simulink.SubsysDataLogs` object named `log`.

tsarray.unpack extracts the time-series objects of class `Simulink.Timeseries` from the `Simulink.TsArray` object named `tsarray`.

**See Also**
`Simulink.ModelDataLogs` | `Simulink.SubsysDataLogs` | `Simulink.Timeseries` | `Simulink.TsArray` | `who` | `whos`

**Topics**
“Export Signal Data Using Signal Logging”

**Introduced before R2006a**
upgradeadvisor

Open Upgrade Advisor

Syntax

upgradeadvisor('modelname')
upgrader = upgradeadvisor('modelname')

Description

upgradeadvisor('modelname') opens the Upgrade Advisor for the model specified by modelname. This command loads the model if necessary, but does not open it in the Simulink Editor. Use the Upgrade Advisor to help you upgrade and improve models with the current release.

upgrader = upgradeadvisor('modelname') returns an object that you can use to analyze and upgrade a hierarchy of models programmatically. If you specify an output, then the Upgrade Advisor does not open. You can use the methods analyze and upgrade with the upgrader object output of the upgradeadvisor function.

• To programmatically analyze a model for recommended upgrades, create an upgrader object and use the method analyze.
• To programmatically analyze and upgrade a model, create an upgrader object and use the method upgrade.
• To configure options before running analyze or upgrade, see “Examples” on page 2-1199.

Tip For an example showing how to programmatically upgrade a whole project, see “Upgrade Simulink Models Using a Project” on page 15-674.
Input Arguments

modelname
Name or handle to the model, specified as a character vector.

Output Arguments

upgrader
Object for analyzing and upgrading the hierarchy of models programmatically.

Examples

Open Upgrade Advisor on a Model

To open the Upgrade Advisor on the vdp example model:

upgradeadvisor('vdp')

To open the Upgrade Advisor on the currently selected model:

upgradeadvisor(bdroot)

Programmatically Analyze and Upgrade a Model

1  Get an upgrader object. This example uses a writable copy of the vdp model.

    load_system('vdp'); save_system('vdp',fullfile(tempdir, 'myvdp'))
    upgrader = upgradeadvisor('myvdp')

    upgrader =

    Upgrader with properties:

        ChecksToSkip: {}
        SkipLibraries: 0
        SkipBlocksets: 1
        OneLevelOnly: 0
To analyze the model for recommended upgrades, following library links and model references, run:

```matlab
analyze(upgrader);
```

You see a report of issues found.

To analyze the model and automatically fix all issues (where automated fixes are available), run:

```matlab
upgrade(upgrader);
```

This command follows library links and model references, and saves any fixes to the model files.

You see a report of issues found and actions taken.

To find the location of the report:

```matlab
reportLocation = upgrader.ReportFile
```

You can configure options before running analyze or upgrade.

- Specify checks to skip before running analyze or upgrade. Find the ID for a check in the Upgrade Advisor by right-clicking the check and selecting **Send Check ID to Workspace**. Then set the `upgrader.ChecksToSkip` property. For example:

  ```matlab
  upgrader.ChecksToSkip = {'mathworks.design.CSStoVSSConvert'};
  upgrade(upgrader);
  ```

- Specify running the Upgrade Advisor on only the current model without following library links or model references:

  ```matlab
  upgrader.OneLevelOnly = true; % default false
  ```

- Specify running the Upgrade Advisor on the current model, following model references but not library links:

  ```matlab
  upgrader.SkipLibraries = true; % default false
  ```

- Specify running the Upgrade Advisor on the current model, including upgrading files in blocksets or toolboxes:

  ```matlab
  upgrader.SkipBlocksets = false; % default true
  ```
By default, the Upgrade Advisor does not upgrade files in blocksets or toolboxes. The Upgrade Advisor detects blocksets from the output of `ver` and the existence of a `Contents.m` file.

- To turn off showing the report after analyze or upgrade, set:

  ```matlab
  upgrader.ShowReport = false; % default true
  ```

**Tip** For an example showing how to programmatically upgrade a whole project, see “Upgrade Simulink Models Using a Project” on page 15-674.

**Tips**

- The Upgrade Advisor can identify cases where you can benefit by changing your model to use new features and settings in Simulink. The Advisor provides advice for transitioning to new technologies, and upgrading a model hierarchy.

  The Upgrade Advisor can also identify cases where a model will not work because changes and improvements in Simulink require changes to a model.

  The Upgrade Advisor offers options to perform recommended actions automatically or instructions for manual fixes.

**Alternatives**

You can also open the Upgrade Advisor from the Simulink Editor. In the **Modeling** tab, select **Model Advisor > Upgrade Advisor**.

Alternatively, you can open the Upgrade Advisor from the Model Advisor. In the Model Advisor, under **By Task checks**, expand the folder **Upgrading to the Current Simulink Version** and select the check **Open the Upgrade Advisor**.

**See Also**

`modeladvisor`

**Topics**

“Consult the Upgrade Advisor”
“Run Model Advisor Checks”
“Upgrade Simulink Models Using a Project” on page 15-674

Introduced in R2012b
**view_mdlrefs**

Display graph of model referencing dependencies without library dependencies

**Syntax**

view_mdlrefs(sys)

**Description**

`view_mdlrefs` opens the “Model Dependency Viewer” with library dependencies omitted. To open a specific configuration of the Model Dependency Viewer, use `depview` with programmatic options. The Model Dependency Viewer provides the same options regardless of how you open it.

`view_mdlrefs(sys)` opens the Model Dependency Viewer, which displays a graph of model referencing dependencies for the specified model. The nodes in the graph represent Simulink models. The directed lines indicate model dependencies.

**Examples**

**Open Model Dependency Viewer with Default Settings**

Open the Model Dependency Viewer for the model `sldemo_mdlref_depgraph`.

`view_mdlrefs('sldemo_mdlref_depgraph');`
Input Arguments

sys — Model name or path
character vector

The full name or path of a model, specified as a character vector.

Data Types: char

See Also

Model | depview | find_mdlrefs

Topics

“Model References”
“Model Dependency Viewer”

Introduced before R2006a
who

List names of top-level data logging objects in Simulink ModelDataLogs data log

Syntax

log.who

tsarray.who

log.who('systems')

log.who('all')

Description

**Note** To list names of top-level data logging objects in Dataset format, use Simulink.SimulationData.Dataset.find.

The ModelDataLogs class is supported for backwards compatibility. Starting in R2016a, you cannot log data in the ModelDataLogs format. Signal logging uses the Dataset format. In R2016a or later, when you open a model from an earlier release that had used ModelDataLogs format, the model simulated in use Dataset format.

You can convert signal logging data from ModelDataLogs to Dataset format. Converting to Dataset format makes it easier to post-process with other logged data (for example, logged states), which can also use Dataset format. For more information, see “Convert Logged Data to Dataset Format”.

If you have legacy code that uses the ModelDataLogs API, you can encounter situations that require updates to your code or model. See “Migrate Scripts That Use Legacy ModelDataLogs API”.

log.who or who(log) lists the names of the top-level signal logging objects contained by log, where log is the handle of a Simulink.ModelDataLogs object name.
tsarray.who or who(tsarray) lists the names of Simulink.TimeSeries objects contained by the Simulink.TsArray object named tsarray.

log.who('systems') or who(log, 'systems') lists the names of all signal logging objects contained by log except for Simulink.Timeseries objects stored in Simulink.TsArray objects contained by log.

log.who('all') or who(log, 'all') lists the names of all the Simulink.Timeseries objects contained by the Simulink.ModelDataLogs, Simulink.TsArray, or Simulink.SubsysDataLogs object named log.

For information about other uses of who, execute help who in the MATLAB Command Window.

**Tip** To get the names of Dataset variables in the MAT-file, using the Simulink.SimulationData.DatasetRef.getDatasetVariableNames function processes faster than using the who or whos functions.

**See Also**

**Topics**
“Load Big Data for Simulations”

**Introduced before R2006a**
**whos**

List names and types of top-level data logging objects in Simulink `ModelDataLogs` data log

**Syntax**

```matlab
log.whos
tsarray.whos
log.whos('systems')
log.whos('all')
```

**Description**

**Note** To list names of top-level data logging objects in `Dataset` format, use `Simulink.SimulationData.Dataset.find`.

The `ModelDataLogs` class is supported for backwards compatibility. Starting in R2016a, you cannot log data in the `ModelDataLogs` format. Signal logging uses the `Dataset` format. In R2016a or later, when you open a model from an earlier release that had used `ModelDataLogs` format, the model simulated in use `Dataset` format.

You can convert signal logging data from `ModelDataLogs` to `Dataset` format. Converting to `Dataset` format makes it easier to post-process with other logged data (for example, logged states), which can also use `Dataset` format. For more information, see “Convert Logged Data to Dataset Format”.

If you have legacy code that uses the `ModelDataLogs` API, you can encounter situations that require updates to your code or model. See “Migrate Scripts That Use Legacy `ModelDataLogs` API”.

```matlab
log.whos` or `whos(log)` lists the names and types of the top-level signal logging objects contained by `log`, where `log` is the handle of a `Simulink.ModelDataLogs` object name.
tsarray.whos or whos(tsarray) lists the names and types of Simulink.TimeSeries objects contained by the Simulink.TsArray object named tsarray.

log.whos('systems') or whos(log, 'systems') lists the names and types of all signal logging objects contained by log except for Simulink.Timeseries objects stored in Simulink.TsArray objects contained by log.

log.whos('all') or whos(log, 'all') lists the names and types of all the Simulink.Timeseries objects contained by the Simulink.ModelDataLogs, Simulink.TsArray or Simulink.SubsysDataLogs object named log.

For information about other uses of who, execute help who in the MATLAB Command Window.

Tip To get the names of Dataset variables in the MAT-file, using the Simulink.SimulationData.DatasetRef.getDatasetVariableNames function processes faster than using the who or whos functions.

See Also

Topics
“Load Big Data for Simulations”

Introduced before R2006a
Mask Icon Drawing Commands

- `color` Change drawing color of subsequent mask icon drawing commands
- `disp` Display text on masked subsystem icon
- `dpoly` Display transfer function on masked subsystem icon
- `droots` Display transfer function on masked subsystem icon
- `fprintf` Display variable text centered on masked subsystem icon
- `image` Display RGB image on masked subsystem icon
- `patch` Draw color patch of specified shape on masked subsystem icon
- `plot` Draw graph connecting series of points on masked subsystem icon
- `port_label` Draw port label on masked subsystem icon
- `text` Display text at specific location on masked subsystem icon
- `block_icon` Promote a block icon image to the masked Subsystem
color

Change drawing color of subsequent mask icon drawing commands

Syntax

color(colorstr)

Here, colorstr must be a character vector.

Description

color(colorstr) sets the drawing color of all subsequent mask drawing commands to the color specified by the string colorstr.

colorstr must be one of the following supported color strings.

blue
green
red
cyan
magenta
yellow
black

Entering any other string or specifying the color using RGB values results in a warning at the MATLAB command prompt; Simulink ignores the color change. The specified drawing color does not influence the color used by the patch or image drawing commands.

Examples

The following commands

    color('cyan');
    droots([-1], [-2 -3], 4)
    color('magenta')
port_label('input',1,'in')
port_label('output',1,'out')

draw the following mask icon.

See Also
droots | port_label

Introduced in R2006b
**disp**

Display text on masked subsystem icon

**Syntax**

\[
\text{disp}(\text{text})
\]

\[
\text{disp}(\text{text}, \text{'texmode', 'on'})
\]

**Description**

\text{disp}(\text{text}) \text{ displays } \text{text} \text{ centered on the block icon. } \text{text} \text{ is any MATLAB expression that evaluates to a string.}

\text{disp(text, 'texmode', 'on')} \text{ allows you to use TeX formatting commands in } \text{text}. \text{ The TeX formatting commands in turn allow you to include symbols and Greek letters in icon text. See "Interpreter" (MATLAB) for information on the TeX formatting commands supported by Simulink software.}

**Examples**

The following command

\[
\text{disp('\{\textbf{Equation:} } \alpha^2 + \beta^2 \rightarrow \gamma^2, \chi, \phi_3 = \{\textbf{cool}\}, \text{'\textarrow{texmode','on'}}\)\}
\]

draws the equation that appears on this masked block icon.
See Also
fprintf | port_label | text

Introduced in R2007a
dpoly

Display transfer function on masked subsystem icon

Syntax

dpoly(num, den)
dpoly(num, den, 'character')

Description

dpoly(num, den) displays the transfer function whose numerator is num and denominator is den.

dpoly(num, den, 'character') specifies the name of the transfer function independent variable. The default is s.

When Simulink draws the block icon, the initialization commands execute and the resulting equation appears on the block icon, as in the following examples:

- To display a continuous transfer function in descending powers of s, enter
dpoly(num, den)

  For example, for num = [0 0 1]; and den = [1 2 1] the icon looks like:

  ![Continuous Transfer Function Icon]

- To display a discrete transfer function in descending powers of z, enter
dpoly(num, den, 'z')

  For example, for num = [0 0 1]; and den = [1 2 1]; the icon looks like:
To display a discrete transfer function in ascending powers of $1/z$, enter

dpoly(num, den, 'z-')

For example, for `num` and `den` as defined previously, the icon looks like:

\[
\frac{1}{z^2+2z+1}
\]

If the parameters are not defined or have no values when you create the icon, Simulink software displays three question marks (?,?,?) in the icon. When you define parameter values in the Mask Settings dialog box, Simulink software evaluates the transfer function and displays the resulting equation in the icon.

**See Also**

disp|droots|port_label|text
**droots**

Display transfer function on masked subsystem icon

**Syntax**

```matlab
droots(zero, pole, gain)
droots(zero, pole, gain, 'z')
droots(zero, pole, gain, 'z-')
```

**Description**

`droots(zero, pole, gain)` displays the transfer function whose zero is `zero`, pole is `pole`, and gain is `gain`.

`droots(zero, pole, gain, 'z')` and `droots(zero, pole, gain, 'z-')` expresses the transfer function in terms of `z` or `1/z`.

When Simulink draws the block icon, the initialization commands execute and the resulting equation appears on the block icon, as in the following examples:

- To display a zero-pole gain transfer function, enter
  ```matlab
droots(z, p, k)
```
  For example, the preceding command creates this icon for these values:
  ```matlab
z = []; p = [-1 -1]; k = 1;
```

If the parameters are not defined or have no values when you create the icon, Simulink software displays three question marks (`??` in the icon. When you define parameter values in the Mask Settings dialog box, Simulink software evaluates the transfer function and displays the resulting equation in the icon.
See Also
disp | dpoly | port_label | text

Introduced in R2007a
fprintf
Display variable text centered on masked subsystem icon

Syntax

fprintf(text)
fprintf(formatSpec, var)

Description

The fprintf command displays formatted text centered on the icon and can display formatSpec along with the contents of var.

Note While this fprintf function is identical in name to its corresponding MATLAB function, it provides only the functionality described on this page.

formatSpec can be a character vector in single quotes, or a string scalar.

Formatting Operator

A formatting operator starts with a percent sign, %, and ends with a conversion character. The conversion character is required. Optionally, you can specify identifier, flags, field width, precision, and subtype operators between % and the conversion character. (Spaces are invalid between operators and are shown here only for readability).

Conversion Character

This table shows conversion characters to format numeric and character data as text.
<table>
<thead>
<tr>
<th>Value Type</th>
<th>Conversion</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer, signed</td>
<td>%d or %i</td>
<td>Base 10</td>
</tr>
<tr>
<td>Integer, unsigned</td>
<td>%u</td>
<td>Base 10</td>
</tr>
<tr>
<td></td>
<td>%o</td>
<td>Base 8 (octal)</td>
</tr>
<tr>
<td></td>
<td>%x</td>
<td>Base 16 (hexadecimal), lowercase letters a–f</td>
</tr>
<tr>
<td></td>
<td>%X</td>
<td>Same as %x, uppercase letters A–F</td>
</tr>
<tr>
<td>Floating-point number</td>
<td>%f</td>
<td>Fixed-point notation (Use a precision</td>
</tr>
<tr>
<td></td>
<td></td>
<td>operator to specify the number of digits</td>
</tr>
<tr>
<td></td>
<td></td>
<td>after the decimal point.)</td>
</tr>
<tr>
<td></td>
<td>%e</td>
<td>Exponential notation, such as 3.141593e+00</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(Use a precision operator to specify the</td>
</tr>
<tr>
<td></td>
<td>%E</td>
<td>number of digits after the decimal point.)</td>
</tr>
<tr>
<td></td>
<td>%g</td>
<td>The more compact of %e or %f, with no</td>
</tr>
<tr>
<td></td>
<td></td>
<td>trailing zeros (Use a precision operator</td>
</tr>
<tr>
<td></td>
<td>%G</td>
<td>to specify the number of significant digits.</td>
</tr>
<tr>
<td>Characters or strings</td>
<td>%c</td>
<td>Single character</td>
</tr>
<tr>
<td></td>
<td>%s</td>
<td>Character vector or string array. The type</td>
</tr>
<tr>
<td></td>
<td></td>
<td>of the output text is the same as the type</td>
</tr>
<tr>
<td></td>
<td></td>
<td>of formatSpec.</td>
</tr>
</tbody>
</table>

**Examples**

The command
fprintf('Hello');
displays the text 'Hello' on the icon.

The command
fprintf('Hello = %d',17);
uses the decimal notation format (%d) to display the variable 17.

See Also
disp | port_label | text

Introduced before R2006a
**image**

Display RGB image on masked subsystem icon

**Syntax**

```matlab
image(a)
image(a, position)
image(a, position, rotation)
```

**Description**

`image(a)` displays the image `a`, where `a` is an \(m\)-by-\(n\)-by-3 array of RGB values. If necessary, use the MATLAB commands `imread` and `ind2rgb` to read and convert bitmap files (such as GIF) to the necessary matrix format.

`image(a, position)` creates the image at the specified position as follows.

<table>
<thead>
<tr>
<th>Position</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>([x, y, w, h])</td>
<td>Position ((x, y)) and size ((w, h)) of the image where the position is relative to the lower-left corner of the mask. The image scales to fit the specified size.</td>
</tr>
<tr>
<td>'center'</td>
<td>Center of the mask</td>
</tr>
<tr>
<td>'top-left'</td>
<td>Top left corner of the mask, unscaled</td>
</tr>
<tr>
<td>'bottom-left'</td>
<td>Bottom left corner of the mask, unscaled</td>
</tr>
<tr>
<td>'top-right'</td>
<td>Top right corner of the mask, unscaled</td>
</tr>
<tr>
<td>'bottom-right'</td>
<td>Bottom right corner of the mask, unscaled</td>
</tr>
</tbody>
</table>

`image(a, position, rotation)` allows you to specify whether the image rotates ('on') or remains stationary ('off') as the icon rotates. The default is 'off'.

**Note** Images in formats `.cur`, `.hdf4`, `.ico`, `.pcx`, `.ras`, `.xwd`, `.svg` (full version) cannot be used as block mask images.
Examples

You can use different commands depending on your requirement to add an image. These commands can be added in the **Icon & Ports** pane of the Mask Editor dialog box.

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>image('icon.jpg')</code></td>
<td>Reads the icon image from a JPEG file named <code>icon.jpg</code> in the MATLAB path.</td>
</tr>
<tr>
<td><code>[data, map]=image('label.gif'); pic=ind2rgb(data,map);</code></td>
<td>Reads and converts a GIF file, <code>label.gif</code>, to the appropriate matrix format.</td>
</tr>
<tr>
<td><code>image(pic)</code></td>
<td>Reads the converted label image.</td>
</tr>
</tbody>
</table>

See Also

`patch` | `plot`

Introduced before R2006a
patch

Draw color patch of specified shape on masked subsystem icon

Syntax

patch(x, y)
patch(x, y, [r g b])

Description

patch(x, y) creates a solid patch having the shape specified by the coordinate vectors x and y. The patch's color is the current foreground color.

patch(x, y, [r g b]) creates a solid patch of the color specified by the vector [r g b], where r is the red component, g the green, and b the blue. For example,

patch([0 .5 1], [0 1 0], [1 0 0])

creates a red triangle on the mask's icon.

Examples

The command

patch([0 .5 1], [0 1 0], [1 0 0])

creates a red triangle on the mask's icon.
See Also

image | plot

Introduced before R2006a
**plot**

Draw graph connecting series of points on masked subsystem icon

**Syntax**

```matlab
plot(Y)
plot(X1,Y1,X2,Y2,...)
```

**Description**

`plot(Y)` plots, for a vector `Y`, each element against its index. If `Y` is a matrix, it plots each column of the matrix as though it were a vector.

`plot(X1,Y1,X2,Y2,...)` plots the vectors `Y1` against `X1`, `Y2` against `X2`, and so on. Vector pairs must be the same length and the list must consist of an even number of vectors.

Plot commands can include `NaN` and `inf` values. When Simulink software encounters `NaNs` or `infs`, it stops drawing, and then begins redrawing at the next numbers that are not `NaN` or `inf`. The appearance of the plot on the icon depends on the units defined by the **Icon units** option in the Mask Editor.

Simulink software displays three question marks (`? ? ?`) in the block icon and issues warnings in these situations:

- When you have not defined values for the parameters used in the drawing commands (for example, when you first create the mask, but have not yet entered values in the Mask Settings dialog box)
- When you enter a masked block parameter or drawing command incorrectly

**Note** The plot command supports all numeric data types.
Examples

The command

\[ \text{plot([0 1 5], [0 0 4])} \]

generates the plot that appears on the icon for the Ramp block, in the Sources library.

See Also

image

Introduced before R2006a
port_label

Draw port label on masked subsystem icon

Syntax

```
port_label('port_type', port_number, 'label')
port_label('port_type', port_number, 'label', 'texmode', 'on')
```

Description

`port_label('port_type', port_number, 'label')` draws a label on a port. Valid values for `port_type` include the following.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>input</td>
<td>Simulink input port</td>
</tr>
<tr>
<td>output</td>
<td>Simulink output port</td>
</tr>
<tr>
<td>lconn</td>
<td>Physical Modeling connection port on the left side of a masked subsystem</td>
</tr>
<tr>
<td>rconn</td>
<td>Physical Modeling connection port on the right side of a masked subsystem</td>
</tr>
<tr>
<td>Enable</td>
<td>Label for the enable port in a masked Triggered or Enabled and Triggered subsystem.</td>
</tr>
<tr>
<td>trigger</td>
<td>Label for the trigger port in a masked Triggered or Enabled and Triggered subsystem.</td>
</tr>
<tr>
<td>action</td>
<td>Label for the action port in a masked Switch Case Action Subsystem.</td>
</tr>
</tbody>
</table>

The input argument `port_number` is an integer, and `label` is text specifying the port's label.
Note  Physical Modeling port labels are assigned based on the nominal port location. If the masked subsystem has been rotated or flipped, for example, a port labeled using 'lconn' as the port_type may not appear on the left side of the block.

\begin{verbatim}
port_label('port_type', port_number, 'label', 'texmode', 'on')
\end{verbatim}

lets you use TeX formatting commands in label. The TeX formatting commands allow you to include symbols and Greek letters in the port label. See “Interpreter” (MATLAB) for information on the TeX formatting commands that the Simulink software supports.

**Examples**

The command

\begin{verbatim}
port_label('input', 1, 'a')
\end{verbatim}

defines a as the label of input port 1.

The command

\begin{verbatim}
port_label('Enable','En')
\end{verbatim}

defines En as the label of Enable port.

The command

\begin{verbatim}
port_label('trigger','Tr')
\end{verbatim}

defines Tr as the label of trigger port.

The command

\begin{verbatim}
port_label('action','Switch():')
\end{verbatim}

defines Switch(): as the label of action port.

The command

\begin{verbatim}
port_label('trigger','$\sqrt m$','interpreter','tex')
\end{verbatim}

defines the label of trigger port with TeX interpretation.

The commands
disp('Card\nSwapper');
port_label('input',1,'\spadesuit','texmode','on');
port_label('output',1,'\heartsuit','texmode','on');

draw playing card symbols as the labels of the ports on a masked subsystem.

See Also
disp | fprintf | text

Introduced before R2006a
text

Display text at specific location on masked subsystem icon

Syntax

text(x, y, 'text')
text(x, y, 'text', 'horizontalAlignment', 'halign',
    'verticalAlignment', 'valign')
text(x, y, 'text', 'texmode', 'on')

Description

The text command places a character vector at a location specified by the point \((x, y)\) whose units are defined by the Icon units option in the Mask Editor.

text(x, y, text, 'texmode', 'on') allows you to use TeX formatting commands in text. The TeX formatting commands in turn allow you to include symbols and Greek letters in icon text. See “Interpreter” (MATLAB) for information on the TeX formatting commands supported by Simulink software.

You can optionally specify the horizontal and/or vertical alignment of the text relative to the point \((x, y)\) in the text command.

The text command offers the following horizontal alignment options.

<table>
<thead>
<tr>
<th>Option</th>
<th>Aligns</th>
</tr>
</thead>
<tbody>
<tr>
<td>'left'</td>
<td>The left end of the text at the specified point</td>
</tr>
<tr>
<td>'right'</td>
<td>The right end of the text at the specified point</td>
</tr>
<tr>
<td>'center'</td>
<td>The center of the text at the specified point</td>
</tr>
</tbody>
</table>

The text command offers the following vertical alignment options.

<table>
<thead>
<tr>
<th>Option</th>
<th>Aligns</th>
</tr>
</thead>
<tbody>
<tr>
<td>'base'</td>
<td>The baseline of the text at the specified point</td>
</tr>
<tr>
<td>Option</td>
<td>Aligns</td>
</tr>
<tr>
<td>---------</td>
<td>-------------------------------------------------------------</td>
</tr>
<tr>
<td>'bottom'</td>
<td>The bottom line of the text at the specified point</td>
</tr>
<tr>
<td>'middle'</td>
<td>The midline of the text at the specified point</td>
</tr>
<tr>
<td>'cap'</td>
<td>The capitals line of the text at the specified point</td>
</tr>
<tr>
<td>'top'</td>
<td>The top of the text at the specified point</td>
</tr>
</tbody>
</table>

**Note** While this text function is identical in name to its corresponding MATLAB function, it provides only the functionality described on this page.

## Examples

### Text Alignment

Center the mask icon text foobar.

```matlab
text(0.5, 0.5, 'foobar', 'horizontalAlignment', 'center')
```

### Equation in Mask Icon

Draw a left-aligned equation as the mask icon.

In the Icons & Ports dialog of the Mask Editor, set Icon units to Normalized.

In the Icon drawing commands text box, enter the following command.

```matlab
text(.05,.5,'
\text{Equation:} \Sigma \alpha^2 + \beta^2 \rightarrow \infty, \Pi, \phi_3 = \textbf{cool}',
'hor','left','texmode','on')
```
Equation: $\Sigma \alpha^2 + \beta^2 \to \infty, \Pi, \phi_3 = \text{cool}$

**See Also**

disp | fprintf | port_label

**Introduced before R2006a**
block_icon

Promote a block icon image to the masked Subsystem

Syntax

block_icon(BlockName)

Description

block_icon(BlockName) displays the underneath block icon image on the masked Subsystem icon. For more information, see slexblockicon.

Input Arguments

BlockName — Name of the underneath block
string

The name of the underneath block whose icon image you want to display on the Subsystem block that encapsulates the specified block. For more information, see slexblockicon

Data Types: string

See Also

Simulink.Mask | slexblockicon

Introduced in R2006b
Simulink Debugger Commands
ashow     Show algebraic loop
atrace    Set algebraic loop trace level
bafter    Insert breakpoint after specified method
break     Insert breakpoint before specified method
bshow     Show specified block
clear     Clear breakpoints from model
continue  Continue simulation
disp      Display block's I/O when simulation stops
ebreak    Enable (or disable) breakpoint on solver errors
elist     List simulation methods in order in which they are executed during simulation
emode     Toggle model execution between accelerated and normal mode
etrace    Enable or disable method tracing
help      Display help for debugger commands
nanbreak  Set or clear nonfinite value break mode
next      Advance simulation to start of next method at current level in model's execution list
probe     I/O and state data for blocks
quit      Stop simulation debugger
rbreak    Break simulation before solver reset
run       Run simulation to completion
slist     Sorted list of model blocks
states    Current state values
status    Debugging options in effect
step      Advance simulation by one or more methods
stimes    Model sample times
stop      Stop simulation
strace    Set solver trace level
systems   List nonvirtual systems of model
tbreak    Set or clear time breakpoint
trace     Display block's I/O each time block executes
undisp    Remove block from debugger's list of display points
untrace   Remove block from debugger's list of trace points
where     Display current location of simulation in simulation loop
xbreak    Break when debugger encounters step-size-limiting state
zcbreak   Toggle breaking at nonsampled zero-crossing events
ashow

Show algebraic loop

Syntax

ashow
ashow gcb
ashow s:b
ashow s#n
ashow clear

Description

ashow returns a list of algebraic loops in the current model.

ashow gcb returns the algebraic loop containing the currently selected block.

ashow s:b returns the algebraic loop containing block b in system s

ashow s#n returns the nth algebraic loop in the system s

ashow clear removes algebraic loop highlighting in the current model

Examples

Highlight algebraic loops during debugging

This example shows you how to find algebraic loops and step through a simulation

Open the sldemo_hydcyl model

Start the Simulink debugger. Select Simulation > Debug > Debug Model.

Run the debugger.
In the MATLAB command prompt, enter:

```
ashow
```

Found 1 Algebraic loop(s):
System number#Algebraic loop id, number of blocks in loop
- 0#1, 9 blocks in loop

The command returns the algebraic loops in the model as well as the number of blocks contained within.

**Find algebraic loops containing selected block**

Open the `sldemo_hydcyl` model and start the Simulink debugger.

Select any nonvirtual block in the model. For more information, see “Nonvirtual and Virtual Blocks”.

For example, select the PlotResults block, which is a Scope block.

In the MATLAB command prompt, enter

```
ashow gcb
```

Block `sldemo_hydcyl/PlotResults` isn't in an algebraic loop.

The PlotResults block does not belong in any algebraic loop.

Look under the mask of the Valve/Cylinder/Piston/Spring Assembly subsystem. To look under the mask, right click on the subsystem and open the context menu. Select **Mask > Look Under Mask**. Alternately, select the subsystem and press `Ctrl+U`.

Select the laminar flow pressure drop block.

In the MATLAB command prompt, enter

```
ashow gcb
```

- `sldemo_hydcyl/Valve//Cylinder//Piston//Spring Assembly/Control Valve Flow/IC`
- `sldemo_hydcyl/Valve//Cylinder//Piston//Spring Assembly/Control Valve Flow/signed sqrt`
- `sldemo_hydcyl/Valve//Cylinder//Piston//Spring Assembly/Control Valve Flow/Product`
- `sldemo_hydcyl/Valve//Cylinder//Piston//Spring Assembly/laminar flow pressure drop`
- `sldemo_hydcyl/Valve//Cylinder//Piston//Spring Assembly/Sum7`
The laminar flow pressure drop block is contained in an algebraic loop with 8 other blocks.

**Input Arguments**

**gcb** — Currently selected block

Currently selected block, specified as gcb.

Example: `ashow gcb`
s:b — Block in model
system index:block index

Block index in the model, specified as system index:block index.
Example: ashow 1:1
Data Types: double

s#n — Algebraic loop index
system index#loop index

Algebraic loop index, specified as system index#loop index
Example: ashow 0#1
Data Types: double

clear — Remove highlighting
clear

Remove algebraic loop highlighting from model.

See Also
atrace

Topics
“Identify Algebraic Loops in Your Model”

Introduced before R2006a
**atrace**

Set algebraic loop trace level

**Syntax**

atrace level

at level

**Arguments**

level Trace level (0 = none, 4 = everything).

**Description**

The `atrace` command sets the algebraic loop trace level for a simulation.

<table>
<thead>
<tr>
<th>Command</th>
<th>Displays for Each Algebraic Loop</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>atrace 0</code></td>
<td>No information</td>
</tr>
<tr>
<td><code>atrace 1</code></td>
<td>The loop variable solution, the number of iterations required to solve the loop, and the estimated solution error</td>
</tr>
<tr>
<td><code>atrace 2</code></td>
<td>Same as level 1</td>
</tr>
<tr>
<td><code>atrace 3</code></td>
<td>Level 2 plus Jacobian matrix used to solve loop</td>
</tr>
<tr>
<td><code>atrace 4</code></td>
<td>Level 3 plus intermediate solutions of the loop variable</td>
</tr>
</tbody>
</table>

**See Also**

`states` | `systems`

*Introduced before R2006a*
**bafter**

Insert breakpoint after specified method

**Syntax**

```plaintext
bafter
bafter m:mid
bafter <sysIdx:blkIdx | gcb> [mth] [tid:TID]
bafter <s:sysIdx | gcs> [mth] [tid:TID]
bafter model [mth] [tid:TID]
```

**Arguments**

- `mid`: Method ID
- `sysIdx:blkId`: Block ID
- `gcb`: Currently selected block
- `sysIdx`: System ID
- `gcs`: Currently selected system
- `model`: Currently selected model
- `mth`: A method name, e.g., `Outputs.Major`
- `TID`: Task ID

**Description**

`bafter` inserts a breakpoint after the current method.

Instead of `bafter`, you can use the short form of `ba` with any of the syntaxes.
bafter **m:mid** inserts a breakpoint after the method specified by **mid** (see “Method ID”).

**bafter sysIdx:blkIdx** inserts a breakpoint after each invocation of the method of the block specified by **sysIdx:blkIdx** (see “Block ID”) in major time steps. **bafter gcb** inserts a breakpoint after each invocation of a method of the currently selected block (see gcb) in major times steps.

**bafter s:sysIdx** inserts a breakpoint after each method of the root system or nonvirtual subsystem specified by the system ID: **sysIdx**.

**Note** The **systems** command displays the system IDs for all nonvirtual systems in the currently selected model.

**bafter gcs** inserts a breakpoint after each method of the currently selected nonvirtual system.

**bafter model** inserts a breakpoint after each method of the currently selected model.

The optional **mth** parameter allow you to set a breakpoint after a particular block, system, or model method and task. For example, **bafter gcb Outputs** sets a breakpoint after the Outputs method of the currently selected block.

The optional TID parameter allows you to set a breakpoint after invocation of a method by a particular task. For example, suppose that the currently selected nonvirtual subsystem operates on task 2 and 3. Then **bafter gcs Outputs tid:2** sets a breakpoint after the invocation of the subsystem’s Outputs method that occurs when task 2 is active.

**See Also**

break | clear | ebreak | nanbreak | rbreak | slist | systems | tbreak | where | xbreak | zcbreak

**Introduced before R2006a**
**break**

Insert breakpoint before specified method

**Syntax**

```
break
b
break m:mid
break <sysIdx:blkIdx | gcb> [mth] [tid:TID]
break <s:sysIdx | gcs> [mth] [tid:TID]
break model [mth] [tid:TID]
```

**Arguments**

- **mid** Method ID
- **sysIdx:blkId** Block ID
- **gcb** Currently selected block
- **sysIdx** System ID
- **gcs** Currently selected system
- **model** Currently selected model
- **mth** A method name, e.g., `Outputs.Major`
- **TID** task ID

**Description**

`break` inserts a breakpoint before the current method.

Instead of `break`, you can use the short form of `b` with any of the syntaxes.
break m:mid inserts a breakpoint before the method specified by mid (see “Method ID”).

break sysIdx:blkIdx inserts a breakpoint before each invocation of the method of the block specified by sysIdx:blkIdx (see “Block ID”) in major time steps. break gcb inserts a breakpoint before each invocation of a method of the currently selected block (see gcb) in major times steps.

break s:sysIdx inserts a breakpoint at each method of the root system or nonvirtual subsystem specified by the system ID: sysIdx.

**Note** The systems command displays the system IDs for all nonvirtual systems in the currently selected model.

break gcs inserts a breakpoint at each method of the currently selected nonvirtual system.

break model inserts a breakpoint at each method of the currently selected model.

**Note** Do not use a model name instead of identifier model in the break model command.

The optional mth parameter allow you to set a breakpoint at a particular block, system, or model method. For example, break gcb Outputs sets a breakpoint at the Outputs method of the currently selected block.

The optional TID parameter allows you to set a breakpoint at the invocation of a method by a particular task. For example, suppose that the currently selected nonvirtual subsystem operates on task 2 and 3. Then break gcs Outputs tid:2 sets a breakpoint at the invocation of the subsystem's Outputs method that occurs when task 2 is active.

**See Also**
bafter | clear | ebreak | nanbreak | rbreak | slist | systems | tbreak | where | xbreak | zcbreak
Introduced before R2006a
bshow

Show specified block

Syntax

bshow s:b
bshow modelName s:b
bs s:b

Arguments

s:b The block whose system index is s and block index is b.

Description

The bshow command opens the model window containing the specified block and selects the block.

See Also

slist

Introduced before R2006a
**clear**

Clear breakpoints from model

**Syntax**

clear

c1

clear m:mid

clear id

clear <sysIdx:blkIdx | gcb>

**Arguments**

-mid Method ID
-id Breakpoint ID
-sysIdx:blk Block ID
-Idx Block ID
-gcb Currently selected block

**Description**

clear clears a breakpoint from the current method.

Instead of clear, you can use the short form of cl with any of the syntaxes.

clear m:mid clears a breakpoint from the method specified by mid.

clear id clears the breakpoint specified by the breakpoint ID id.

clear sysIdx:blkIdx clears any breakpoints set on the methods of the block specified by sysIdx:blkIdx.
clear gcb clears any breakpoints set on the methods of the currently selected block.

See Also
bafter | break | slist

Introduced before R2006a
continue

Continue simulation

Syntax

continue
c

Description

The `continue` command continues the simulation from the current breakpoint. If animation mode is not enabled, the simulation continues until it reaches another breakpoint or its final time step. If animation mode is enabled, the simulation continues in animation mode to the first method of the next major time step, ignoring breakpoints.

See Also

quit | run | stop

Introduced before R2006a
**disp**

Display block's I/O when simulation stops

**Syntax**

```matlab
disp
disp gcb
disp s:b
disp modelName s:b
```

**Arguments**

- `s:b` The block whose system index is `s` and block index is `b`.
- `gcb` Current block.

**Description**

The `disp` command registers a block as a display point. The debugger displays the inputs and outputs of all display points in the MATLAB Command Window whenever the simulation halts. Invoking `disp` without arguments shows a list of display points. Use `undisp` to unregister a block.

Instead of `disp`, you can use the short form of `d` with any of the syntaxes.

**See Also**

`probe` | `slist` | `trace` | `undisp`

**Introduced before R2006a**
ebreak

Enable (or disable) breakpoint on solver errors

Syntax

ebreak
eb

Description

This command causes the simulation to stop if the solver detects a recoverable error in
the model. If you do not set or disable this breakpoint, the solver recovers from the error
and proceeds with the simulation without notifying you.

See Also

bafter | break | clear | nanbreak | rbreak | slist | systems | tbreak | where |
xbreak | zcbreak

Introduced before R2006a
elist

List simulation methods in order in which they are executed during simulation

Syntax

eлист

eлист

eлист m:mid [tid:TID]
eлист <gcs | s:sysIdx> [mth] [tid:TID]
eлист <gcb | sysIdx:blkIdx> [mth] [tid:TID]

Description

Instead of elist, you can use the short form of el with any of the syntaxes.

elist m:mid lists the methods invoked by the system or nonvirtual subsystem method corresponding to the method id mid (see the where command for information on method IDs), e.g.,

```
(slddebug @19): elist m:19
```

```
RootSystem.Outputs 'vdp' [tid=0] :
  0:0 Integrator Outputs 'vdp/x1' [tid=0]
  0:1 Outport Outputs 'vdp/Out1' [tid=0]
  0:2 Integrator Outputs 'vdp/x2' [tid=0]
  ... Block id Method Block Task id
```

The method list specifies the calling method followed by the methods that it calls in the order in which they are invoked. The entry for the calling method includes
• The name of the method

The name of the method is prefixed by the type of system that defines the method, e.g., RootSystem.
• The name of the model or subsystem instance on which the method is invoked
• The ID of the task that invokes the method

The entry for each called method includes

• The ID (sysIdx:blockIdx) of the block instance on which the method is invoked

The block ID is prefixed by a number specifying the system that contains the block (the sysIdx). This allows Simulink software to assign the same block ID to blocks residing in different subsystems.
• The name of the method

The method name is prefixed with the type of block that defines the method, e.g., Integrator.
• The name of the block instance on which the method is invoked
• The task that invokes the method

The optional task ID parameter (tid:TID) allows you to restrict the displayed lists to methods invoked for a specified task. You can specify this option only for system or atomic subsystem methods that invoke Outputs or Update methods.

`elist <gcs | s:sysIdx>` lists the methods executed for the currently selected system (specified by the gcs command) or the system or nonvirtual subsystem specified by the system ID sysIdx, e.g.,
The system ID of a model's root system is 0. You can use the debugger's `systems` command to determine the system IDs of a model's subsystems.

**Note** The `elist` and `where` commands use block IDs to identify subsystems in their output. The block ID for a subsystem is not the same as the system ID displayed by the `systems` command. Use the `elist sysIdx:blkIdx` form of the `elist` command to display the methods of a subsystem whose block ID appears in the output of a previous invocation of the `elist` or `where` command.

```
elist <gcs | s:sysIdx> mth  lists methods of type `mth` to be executed for the system specified by the `gcs` command or the system ID `sysIdx`, e.g.,
```
Use \texttt{elist gcb} to list the methods invoked by the nonvirtual subsystem currently selected in the model.

\textbf{See Also}
\texttt{slist} | \texttt{systems} | \texttt{where}

\textit{Introduced before R2006a}
emode

Toggle model execution between accelerated and normal mode

Syntax

dmode

dm

Description

Toggles the simulation between accelerated and normal mode when using the Accelerator mode in Simulink software. For more information, see “Run Accelerator Mode with the Simulink Debugger”.

Introduced before R2006a
ettrace

Enable or disable method tracing

Syntax

ettrace level level-number
et level level-number

Description

This command enables or disables method tracing, depending on the value of level:

<table>
<thead>
<tr>
<th>Level</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Turn tracing off.</td>
</tr>
<tr>
<td>1</td>
<td>Trace model methods.</td>
</tr>
<tr>
<td>2</td>
<td>Trace model and system methods.</td>
</tr>
<tr>
<td>3</td>
<td>Trace model, system, and block methods.</td>
</tr>
</tbody>
</table>

When method tracing is on, the debugger prints a message at the command line every time a method of the specified level is entered or exited. The message specifies the current simulation time, whether the simulation is entering or exiting the method, the method id and name, and the name of the model, system, or block to which the method belongs.

See Also

elist | trace | where

Introduced before R2006a
help
Display help for debugger commands

Syntax
help
?
h

Description
The help command displays a list of debugger commands in the command window. The list includes the syntax and a brief description of each command.

Introduced before R2006a
nanbreak

Set or clear nonfinite value break mode

Syntax
nanbreak
na

Description
The nanbreak command causes the debugger to break whenever the simulation encounters a nonfinite (NaN or Inf) value. If nonfinite break mode is set, nanbreak clears it.

See Also
bafter | break | rbreak | tbreak | xbreak | zcbreak

Topics
ebreak

Introduced before R2006a
**next**

Advance simulation to start of next method at current level in model's execution list

**Syntax**

next

n

**Description**

The `next` command advances the simulation to the start of the next method at the current level in the model's method execution list.

**Note** The `next` command has the same effect as the `step` `over` command. See `step` for more information.

**See Also**

step

*Introduced before R2006a*
probe

I/O and state data for blocks

Syntax

probe
probe s:b
probe modelName s:b
probe gcb
probe level level-type
p

Description

probe sets the Simulink debugger to interactive probe mode. In this mode, the debugger displays the I/O of a selected block. To exit interactive probe mode, enter a debugger command or press the Enter key.

probe s:b displays the I/O of the block whose system index is s and block index is b.

probe modelName s:b displays the I/O of the block system index is s and block index is b in the modelName model.

probe gcb displays the I/O of the currently selected block.

probe level level-type sets the verbosity level for probe, trace, and dis. If level-type is io, the debugger displays block I/O. If level-type is all (default), the debugger displays all information for the current state of a block, including inputs and outputs, states, and zero crossings.

p is the short form of the command.
Examples

Display I/O for the currently selected block `Out2` in the model `vdp` using the Simulink debugger.

1 In the MATLAB Command Window, enter:

   `sldebug 'vdp'

   The MATLAB command prompt `>>` changes to the Simulink debugger prompt `(sldebug @0): >>`.

2 Enter:

   `probe gcb`

   The MATLAB Command Window displays:

   ```
   probe: Data of 0:3 Outport block 'vdp/Out2':
   U1     = [0]
   ```

See Also

`disp` | `trace`

Introduced in R2007a
quit

Stop simulation debugger

Syntax

quit
q

Description

quit stops the Simulink debugger and returns to the MATLAB command prompt.
q is the short form of the command.

Examples

Start the Simulink debugger for the model vdp and then stop it.
1 In the MATLAB Command Window, enter:
   sldebug 'vdp'
   The MATLAB command prompt >> changes to the Simulink debugger prompt (sldebug @0): >>.
2 Enter:
   quit

See Also

stop

Introduced before R2006a
rbreak

Break simulation before solver reset

Syntax

rbreak
rb

Description

rbreak enables (or disables) a solver reset breakpoint if the breakpoint is disabled (or enabled). The breakpoint causes the debugger to halt the simulation whenever an event requires a solver reset. The halt occurs before the solver resets.

rb is the short form of the command.

Examples

Start Simulink debugger for the model vdp and a set breakpoint before a solver reset.

1 In the MATLAB Command Window, enter:

    sldebug 'vdp'

The MATLAB command prompt >> is replaced with the Simulink debugger prompt (sldebug @0): >>.

2 Enter:

    rbreak

The MATLAB Command Window displays:

    Break on solver reset request : enabled
See Also
bafter | break | ebreak | nanbreak | tbreak | xbreak | zcbreak

Introduced before R2006a
run

Run simulation to completion

Syntax

run
r

Description

run starts the simulation from the current breakpoint to its final time step. It ignores breakpoints and display points.

r is the short form of the command

Examples

Continue the simulation for the model vdp using the Simulink debugger.

1 In the MATLAB Command Window, enter:
   
   sldebug 'vdp'
   
   The MATLAB command prompt >> changes to the Simulink debugger prompt (sldebug @0): >>.

2 Enter:

run

See Also

continue | quit | stop
Introduced before R2006a
**slist**

Sorted list of model blocks

**Syntax**

```
sl
sl
sli
```

**Description**

`slist` displays a list of blocks for the root system and each nonvirtual subsystem sorted according to data dependencies and other criteria.

For each system (root or nonvirtual), `slist` displays:

- Title line specifying the name of the system, the number of nonvirtual blocks that the system contains, and the number of blocks in the system that have direct feedthrough ports.
- Entry for each block in the order in which the blocks appear in the sorted list.

For each block entry, `slist` displays the block ID and the name and type of the block. The block ID consists of a system index and a block index separated by a colon (`sysIdx:blkIdx`).

- Block index is the position of the block in the sorted list.
- System index is the order in which the Simulink software generated the system sorted list. The system index has no special significance. It simply allows blocks that appear in the same position in different sorted lists to have unique identifiers.

Simulink software uses sorted lists to create block method execution lists (see `elist`) for root system and nonvirtual subsystem methods. In general, root system and nonvirtual subsystem methods invoke the block methods in the same order as the blocks appear in the sorted list.

Exceptions occur in the execution order of block methods. For example, execution lists for multicast models group together all blocks operating at the same rate and in the same
task. Slower groups appear later than faster groups. The grouping of methods by task can result in a block method execution order that is different from the block sorted order. However, within groups, methods execute in the same order as the corresponding blocks appear in the sorted list.

sli is the short form of the command.

**Examples**

Display a sorted list of the root system in the vdp model using the Simulink debugger.

1. In the MATLAB Command Window, enter:

```
sldebug 'vdp'
```

The MATLAB command prompt `>>` changes to the Simulink debugger prompt (`sldebug @0): >>`.

2. Enter:

```
slist
```

The MATLAB Command Window displays:

```
---- Sorted list for 'vdp' [9 nonvirtual blocks, directFeed=0]
0:0  'vdp/x1' (Integrator)
0:1  'vdp/Out1' (Outport)
0:2  'vdp/x2' (Integrator)
0:3  'vdp/Out2' (Outport)
0:4  'vdp/Scope' (Scope)
0:5  'vdp/Fcn' (Fcn)
0:6  'vdp/Product' (Product)
0:7  'vdp/Mu' (Gain)
0:8  'vdp/Sum' (Sum)
```

**See Also**

elist | systems

**Introduced before R2006a**
states

Current state values

Syntax

states

Description

states displays a list of the current states of the model. The list includes the index, current value, system:block:element ID, state vector name, and block name for each state.

Examples

Display information about the states for the vdp model.

1. In the MATLAB Command Window, enter:

   sldebug 'vdp'

   The MATLAB command prompt >> changes to the Simulink debugger prompt (sldebug @0): >>.

2. Enter:

   states

   The MATLAB Command Window displays:

   Continuous States:
  Idx Value (system:block:element Name 'BlockName')
  0  0 (0:0:0 CSTATE 'vdp/x1')
  1  0 (0:2:0 CSTATE 'vdp/x2')
status

Debugging options in effect

Syntax

Description

Display a list of the debugging options in effect.

Examples

View Debugger Status for vdp

Start the debugger with vdp. The command prompt changes to the Simulink debugger prompt (sldebug @0): >>.

sldebug 'vdp'

Display the debugging status.

status

%-----------------------------------------------
Current simulation time : 0 (MajorTimeStep)
Solver needs reset : no
Solver derivatives cache needs reset : no
Zero crossing signals cache needs reset : no
Default command to execute on return/enter : ""
Break at zero crossing events : disabled
Break on solver error : disabled
Break on failed integration step : disabled
Time break point : disabled
Break on non-finite (NaN,Inf) values : disabled
Break on solver reset request : disabled
Display level for disp, trace, probe : 1 (i/o, states)
Solver trace level : 0
Algebraic loop tracing level : 0
Animation Mode : off
Execution Mode : Normal
Display level for etrace : 0 (disabled)
Break points : none installed
Display points : none installed

Introduced before R2006a
**step**

Advance simulation by one or more methods

**Syntax**

```
step
step in
step over
step out
step top
step blockmth
s
```

**Description**

- `step` or `step in` advances the simulation to the next method in the current time step.
- `step over` advances the simulation over the next method.
- `step out` advances the simulation to the end of the current simulation point hierarchy.
- `step top` advances the simulation to the first method executed in the next time step.
- `step blockmth` advances the simulation to the next method that operates on a block.
- `s` is the short form of the command.

If `step` advances the simulation to the start of a block method, the debugger points at the block on which the method operates.
Examples

The following diagram illustrates the effect of various forms of the step command for the model vdp.

See Also
elist | next | where

Introduced in R2007a
**stimes**

Model sample times

**Syntax**

```
stimes  
sti  
```

**Description**

`stimes` displays information about the model sample times, including the sample time period, offset, and task ID.

`sti` is the short form of the command.

**Examples**

Display sample times for the model `vdp` using the Simulink debugger.

1. In the MATLAB Command Window, enter:
   ```
   sldebug 'vdp'
   
   The MATLAB command prompt `>>` changes to the Simulink debugger prompt `(sldebug @0): >>`.
   
2. Enter:
   ```
   stimes
   
   The MATLAB Command Window displays:
   
   --- Sample times for 'vdp' [Number of sample times = 1]  
   1. [0     , 0     ] tid=0 (continuous sample time)
Introduced before R2006a
stop
Stop simulation

Syntax
stop

Description
stop stops the simulation of the model you are debugging.

Examples
Start and stop a simulation for the model vdp using the Simulink debugger.

1. Start a debugger session. In the MATLAB Command Window, enter:
   
   sldebug 'vdp'

   The MATLAB command prompt >> changes to the Simulink debugger prompt (sldebug @0): >>.

2. Start a simulation of the model. Enter:
   
   run

3. Stop the simulation. Enter:
   
   stop

See Also
continue | quit | run

Introduced before R2006a
strace

Set solver trace level

Syntax

strace level
i

Description

strace level causes the solver to display diagnostic information in the MATLAB Command Window, depending on the value of level. Values are 0 (no information) or 1 (maximum information about time steps, integration steps, zero crossings, and solver resets).

i is the short form of the command.

Examples

Display maximum information about a simulation for the model vdp using the Simulink debugger.

1. In the MATLAB Command Window, enter:

   sldebug 'vdp'

   The MATLAB command prompt >> changes to the Simulink debugger prompt (sldebug @0): >>.

2. Get information about the notation . Enter:

   help time

   The MATLAB Command Window displays:

   Time is displayed as:
   TM = <time while in MajorTimeStep>
Set trace to display all information. Enter:

strace 1

When diagnostic tracing is on, the debugger displays the sizes of major and minor time steps.

[TM = 13.21072088374186 ] Start of Major Time Step
[Tm = 13.21072088374186 ] Start of Minor Time Step

The debugger displays integration information. This information includes the time step of the integration method, step size of the integration method, outcome of the integration step, normalized error, and index of the state.

[Tm = 13.21072088374186 ] [H = 0.2751116230148764 ] Begin Integration Step
[Tf = 13.48583250675674 ] [Hf = 0.2751116230148764 ] Fail [Er = 1.0404e+000 ] [Ix = 1 ]
[Tm = 13.21072088374186 ] [H = 0.2183536061326544 ] Retry
[Ts = 13.42907448987452 ] [Hs = 0.2183536061326539 ] Pass [Er = 2.8856e-001 ] [Ix = 1 ]

For zero crossings, the debugger displays information about the iterative search algorithm when the zero crossing occurred. This information includes the time step of the zero crossing, step size of the zero crossing detection algorithm, length of the time interval bracketing the zero crossing, and a flag denoting the rising or falling direction of the zero crossing.

[Tz = 3.615333333333301 ] Detected 1 Zero Crossing Event 0[F]
Begin iterative search to bracket zero crossing event
[Tz = 3.621111157580072 ] [Hz = 0.005777824246771424 ] [Iz = 4.2222e-003 ] 0[F]
[Tz = 3.621119820808098 ] [Hz = 0.005783648746797265 ] [Iz = 4.2164e-003 ] 0[F]
[Tz = 3.62111987943544 ] [Hz = 0.005783654610242994 ] [Iz = 4.2163e-003 ] 0[F]
[Tz = 3.62111987943544 ] [Hz = 0.005783654610242994 ] [Iz = 1.1804e-011 ] 0[F]
[Tz = 3.62111987949452 ] [Hz = 0.005783654616151157 ] [Iz = 5.8962e-012 ] 0[F]
[Tz = 3.62111987949452 ] [Hz = 0.005783654616151157 ] [Iz = 5.1514e-014 ] 0[F]
End iterative search to bracket zero crossing event
When a solver resets occur, the debugger displays the time at which the solver was reset.

\[ \text{Tr} = 6.246905153573676 \] Process Solver Reset
\[ \text{Tr} = 6.246905153573676 \] Reset Zero Crossing Cache
\[ \text{Tr} = 6.246905153573676 \] Reset Derivative Cache

**See Also**

atrace | etrace | states | trace | zclist

**Introduced before R2006a**
systems

List nonvirtual systems of model

Syntax

systems
sys

Description

systems displays the nonvirtual subsystems for a model in the MATLAB Command Window.

sys is the short form of the command.

Examples

Display the nonvirtual systems for the model sldemo_enginewc using the Simulink debugger.

1 In the MATLAB Command Window, enter:
   sldebug 'sldemo_enginewc'

   The MATLAB command prompt >> changes to the Simulink debugger prompt (sldebug @0): >>.

2 Enter:
   systems

   The MATLAB Command Window displays the nonvirtual subsystems.
   
   0  'sldemo_enginewc'
   2  'sldemo_enginewc/Compression'
   3  'sldemo_enginewc/Controller'
   4  'sldemo_enginewc/Throttle & Manifold/Throttle/TmpAtomicSubsysAtSwitchInport1'
   5  'sldemo_enginewc/valve timing/positive edge to dual edge conversion'
See Also
slist

Introduced before R2006a
**tbreak**

Set or clear time breakpoint

**Syntax**

tbreak
tb
tbreak t

**Description**

The `tbreak` command sets a breakpoint at the specified time step. If a breakpoint already exists at the specified time, `tbreak` clears the breakpoint. If you do not specify a time, `tbreak` toggles a breakpoint at the current time step.

Instead of `tbreak`, you can use the short form of `tb`, with or without `t`.

**See Also**

`bafter` | `break` | `nanbreak` | `rbreak` | `xbreak` | `zcbreak`

**Topics**

`ebreak`

**Introduced before R2006a**
trace

Display block's I/O each time block executes

Syntax

trace gcb
trace s:b

tr gcb
trace s:b

Arguments

s:b The block whose system index is s and block index is b.
gcb   Current block.

Description

The trace command registers a block as a trace point. The debugger displays the I/O of each registered block each time the block executes.

See Also

disp|probe|slist|strace|untrace

Introduced before R2006a
undisp

Remove block from debugger's list of display points

Syntax
undisp gcb
und gcb
undisp s:b
und s:b

Arguments
s:b    The block whose system index is s and block index is b.
gcb    Current block.

Description
The undisp command removes the specified block from the debugger's list of display points.

See Also
disp|slist

Introduced before R2006a
untrace

Remove block from debugger's list of trace points

Syntax

untrace gcb
unt gcb
untrace s:b
unt s:b

Arguments

s:b     The block whose system index is s and block index is b.
gcb    Current block.

Description

The untrace command removes the specified block from the debugger's list of trace points.

See Also

slist | trace

Introduced before R2006a
**where**

Display current location of simulation in simulation loop

**Syntax**

```
where [detail]
w [detail]
```

**Description**

The `where` command displays the current location of the simulation in the simulation loop, for example,

```
sldebug @7): where
  0 >> vdp.Simulate
  1   >> vdp.Start
  2     >> RootSystem.Start 'vdp'
  7   >| 0:8 Sum.Start 'Sum'
```

The display consists of a list of simulation nodes with the last entry being the node that is about to be entered or exited. Each entry contains the following information:

- **Method ID**
  
  The method ID identifies a specific invocation of a method.

- **A symbol specifying its state:**
  
  - `>>` (active)
  - `>|` (about to be entered)
  - `<|` (about to be exited)
- Name of the method invoked (e.g., RootSystem.Start)
- Name of the block or system on which the method is invoked (e.g., Sum)
- System and block ID (sysIdx:blkIdx) of the block on which the method is invoked

For example, 0:8 indicates that the specified method operates on block 8 of system 0.

where detail, where detail is any nonnegative integer, includes inactive nodes in the display.

```plaintext
0 >> vdp.Simulate
   1   >> vdp.Start
   2   >> RootSystem.Start 'vdp'
   3     0:4 Scope.Start 'Scope'
   4     0:5 Fcn.Start 'Fcn'
   5     0:6 Product.Start
   'Product'
   6     0:7 Gain.Start 'Mu'
   7     >| 0:8 Sum.Start 'Sum'
```

See Also

step

Introduced before R2006a
**xbreak**

Break when debugger encounters step-size-limiting state

**Syntax**

xbreak
x

**Description**

The `xbreak` command pauses execution of the model when the debugger encounters a state that limits the size of the steps that the solver takes. If `xbreak` mode is already on, `xbreak` turns the mode off.

**See Also**

`bafter` | `break` | `nanbreak` | `rbreak` | `tbreak` | `zcbreak`

**Topics**

ebreak

Introduced before R2006a
zcbreak

Toggle breaking at nonsampled zero-crossing events

Syntax
zcbreak
zc

Description
The zcbreak command causes the debugger to break when a nonsampled zero-crossing event occurs. If zero-crossing break mode is already on, zcbreak turns the mode off.

See Also
bafter | break | nanbreak | tbreak | xbreak | zclist

Introduced before R2006a
**zclist**

List blocks containing nonsampled zero crossings

**Syntax**

```matlab
zclist
zcl
```

**Description**

The `zclist` command displays a list of blocks in which nonsampled zero crossings can occur. The command displays the list in the MATLAB Command Window.

**See Also**

`zcbreak`

**Introduced before R2006a**
Simulink Classes
<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>eventData</td>
<td>Provide information about block method execution events</td>
</tr>
<tr>
<td>Simulink.Annotation</td>
<td>Specify properties of model annotation</td>
</tr>
<tr>
<td>Simulink.BlockCompDworkData</td>
<td>Provide postcompilation information about block's DWork vector</td>
</tr>
<tr>
<td>Simulink.BlockCompInputPortData</td>
<td>Provide postcompilation information about block input port</td>
</tr>
<tr>
<td>Simulink.BlockCompOutputPortData</td>
<td>Provide postcompilation information about block output port</td>
</tr>
<tr>
<td>Simulink.BlockData</td>
<td>Provide run-time information about block-related data, such as block parameters</td>
</tr>
<tr>
<td>Simulink.BlockPath</td>
<td>Fully specified Simulink block path</td>
</tr>
<tr>
<td>Simulink.BlockPortData</td>
<td>Describe block input or output port</td>
</tr>
<tr>
<td>Simulink.BlockPreCompInputPortData</td>
<td>Provide precompilation information about block input port</td>
</tr>
<tr>
<td>Simulink.BlockPreCompOutputPortData</td>
<td>Provide precompilation information about block output port</td>
</tr>
<tr>
<td>Simulink.ConfigSetRef</td>
<td>Link model to configuration set stored independently of any model</td>
</tr>
<tr>
<td>Simulink.GlobalDataTransfer</td>
<td>Configure concurrent execution data transfers</td>
</tr>
<tr>
<td>Simulink.MDLInfo</td>
<td>Extract model file information without loading block diagram into memory</td>
</tr>
<tr>
<td>getDescription</td>
<td>Extract model file description without loading block diagram into memory</td>
</tr>
<tr>
<td>getMetadata</td>
<td>Extract model file metadata without loading block diagram into memory</td>
</tr>
<tr>
<td>Simulink.ModelDataLogs</td>
<td>Container for signal data logs of a model</td>
</tr>
<tr>
<td>Simulink.SimState.ModelSimState</td>
<td>(Not recommended) Access SimState snapshot data</td>
</tr>
<tr>
<td>Simulink.MSFcnRunTimeBlock</td>
<td>Get run-time information about Level-2 MATLAB S-function block</td>
</tr>
<tr>
<td>Simulink.RunTimeBlock</td>
<td>Allow Level-2 MATLAB S-function and other MATLAB programs to get information about block while simulation is running</td>
</tr>
</tbody>
</table>
matlab.io.datastore.sdiddatastore class

Package: matlab.io.datastore

Datastore for Simulation Data Inspector signals

Description

A `matlab.io.datastore.sdiddatastore` object provides access to signals logged to
the Simulation Data Inspector that are too large to fit into memory. An `sdiddatastore`
object references the data for a single signal. The `read` method loads the signal data
referenced by an `sdiddatastore` object in a chunk-wise manner such that each chunk
always fits into memory. You can use an `sdiddatastore` object to create a tall timetable
for your signal data. For more information about working with tall arrays, see “Tall
Arrays” (MATLAB).

**Note** `matlab.io.datastore.sdiddatastore` does not support parallel computations.
If you have a Parallel Computing Toolbox license, use `mapreducer(0)` to set the
execution environment to the local MATLAB client before creating a tall timetable from
the `matlab.io.datastore.sdiddatastore`.

Construction

```matlab
ds = dsrObj.getAsDatastore(arg) creates the `sdiddatastore`, `ds`, for the signal in
the Simulink.sdi.DatasetRef object selected by the search criterion `arg`. You can
specify `arg` as an integer representing the index of the desired signal within the
Simulink.sdi.DatasetRef object, or as a character vector containing the name of the
signal.

ds = matlab.io.datastore.sdiddatastore(signalID) creates the `sdiddatastore`, `ds`, for the signal corresponding to the specified `signalID`.
```

Input Arguments

`arg` — Element selection criterion
can be either a character vector or an integer.
Search criterion used to retrieve the element from the Simulink.sdi.DatasetRef object. For name-based searches, specify arg as a character vector. For index-based searches, arg is an integer, representing the index of the desired element.

Example: 'MySignal'
Example: 3

**signalID** — Numeric signal identifier

integer

Numeric signal identifier generated for the signal by the Simulation Data Inspector. You can get the signal ID for a signal using methods of the Simulink.sdi.Run object containing the signal or as a return from the Simulink.sdi.createRun function.

**Properties**

**Name** — Signal name

character vector

Name of the signal specified as a character vector.

Example: 'My Signal'

**Signal** — Simulink.sdi.Signal object

Simulink.sdi.Signal object associated with the sdidatastore. The Signal property provides access to the signal data and metadata.

**Methods**

- hasdata: Determine if data is available to read
- preview: Return preview of data in sdidatastore
- read: Read a chunk of data from an sdidatastore
- readall: Read all data from an sdidatastore
- reset: Reset the read position
Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects (MATLAB).

Examples

Create a Tall Timetable for a Simulation Data Inspector Signal

This example shows how to create a tall timetable for a signal in the Simulation Data Inspector repository. You can create the tall timetable using a Simulink.sdi.Signal object or by first creating a matlab.io.datastore.sdidatastore for the signal. You can use a matlab.io.datastore.sdidatastore to incrementally read and process signal data for signals that do not fit into memory. A tall timetable handles the data chunking and processing in the background. In general, you can work with tall timetables very similarly to how you work with in-memory data.

Create Data and Access Signal ID

Whether you create your tall timetable using a Simulink.sdi.Signal object or a matlab.io.datastore.sdidatastore, start by creating data and accessing the signal ID for a signal of interest. The sldemo_fuelsys model is configured to log signals which stream to the Simulation Data Inspector repository when you simulate the model.

```matlab
open_system('sldemo_fuelsys')
sim('sldemo_fuelsys')
```

Then, use the Simulation Data Inspector programmatic interface to access the signal ID for a signal of interest. For example, access the ego signal.

```matlab
runCount = Simulink.sdi.getRunCount;
latestRunID = Simulink.sdi.getRunIDByIndex(runCount);
latestRun = Simulink.sdi.getRun(latestRunID);

egoSigID = latestRun.getSignalIDByIndex(7);
```

Create a Tall Timetable Using a matlab.io.datastore.sdidatastore

In general, tall timetables are backed by datastores. Create a matlab.io.datastore.sdidatastore object to reference the signal data in the Simulation Data Inspector repository.
egoDs = matlab.io.datastore.sdidatastore(egoSigID);

Check the name of the datastore to verify you have the signal you expect.

egoDs.Name

ans =
'fuel'

Create a tall timetable from the `matlab.io.datastore.sdidatastore` to use for processing the signal data. When you have a Parallel Computing Toolbox™ license, you need to explicitly set the execution environment to the local MATLAB® session using `mapreducer` before creating the tall timetable. The `matlab.io.datastore.sdidatastore` object does not support parallel computations.

mapreducer(0);

egoTt = tall(egoDs)

egoTt =

Mx1 tall timetable

<table>
<thead>
<tr>
<th>Time</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 sec</td>
<td>1.209</td>
</tr>
<tr>
<td>0.00056199 sec</td>
<td>1.209</td>
</tr>
<tr>
<td>0.0033719 sec</td>
<td>1.209</td>
</tr>
<tr>
<td>0.01 sec</td>
<td>1.1729</td>
</tr>
<tr>
<td>0.02 sec</td>
<td>1.1409</td>
</tr>
<tr>
<td>0.03 sec</td>
<td>1.1124</td>
</tr>
<tr>
<td>0.04 sec</td>
<td>1.0873</td>
</tr>
<tr>
<td>0.05 sec</td>
<td>1.0652</td>
</tr>
<tr>
<td>:</td>
<td>:</td>
</tr>
</tbody>
</table>

Create a Tall Timetable Using a Simulink.sdi.Signal Object

The `Simulink.sdi.Signal` class has a method to create a tall timetable directly, allowing you to skip the step of creating a datastore by creating it behind the scenes. Use the signal ID to access the `Simulink.sdi.Signal` object for the ego signal. Then, use the `getTable` method to create the tall timetable.
egoSig = Simulink.sdi.getSignal(egoSigID);

egoTt = egoSig.getAsTall

egoTt =

Mx1 tall timetable

<table>
<thead>
<tr>
<th>Time</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 sec</td>
<td>1.209</td>
</tr>
<tr>
<td>0.00056199 sec</td>
<td>1.209</td>
</tr>
<tr>
<td>0.0033719 sec</td>
<td>1.209</td>
</tr>
<tr>
<td>0.01 sec</td>
<td>1.1729</td>
</tr>
<tr>
<td>0.02 sec</td>
<td>1.1409</td>
</tr>
<tr>
<td>0.03 sec</td>
<td>1.1124</td>
</tr>
<tr>
<td>0.04 sec</td>
<td>1.0873</td>
</tr>
<tr>
<td>0.05 sec</td>
<td>1.0652</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Use the Tall Timetable to Process Your Signal Data

When you use the tall timetable egoTt, its underlying datastore reads chunks of data and passes them to the tall timetable to process. Neither the datastore nor the tall timetable retain any of the data in memory after processing. Also, the tall timetable defers processing for many operations. For example, calculate the mean value of the signal.

egoMean = mean(egoTt.Data)

egoMean =

tall double

You can use the gather function to evaluate a variable and write its value to the workspace, or you can use the write function to write the results to disc. When you use gather, be sure the results fit into memory.

egoMean = gather(egoMean)
Evaluating tall expression using the Local MATLAB Session:
- Pass 1 of 1: Completed in 2 min 51 sec
Evaluation completed in 2 min 51 sec

goingMean = 1.3292

When you perform multiple operations on a tall timetable, evaluation of the results for each step is deferred until you explicitly request the results with write or gather. During evaluation, MATLAB optimizes the number of passes it makes through the tall timetable, which can significantly speed up processing time for analyzing very large signals. For more information about working with tall arrays, see “Tall Arrays for Out-of-Memory Data” (MATLAB).

**Process Signal Data Using a `matlab.io.datastore.sdidatastore`**

A `matlab.io.datastore.sdidatastore` references signal data in the Simulation Data Inspector repository. When the signal is too large to fit into memory, you can use the `matlab.io.datastore.sdidatastore` to incrementally process the data manually or to create a tall timetable for the signal that handles the incremental processing for you. This example shows how to process data using a `matlab.io.datastore.sdidatastore`.

**Create a `matlab.io.datastore.sdidatastore` for a Signal**

Simulate the `sldemo_fuelsys` model, which is configured to log several signals, to create data in the Simulation Data Inspector repository.

```matlab
sim('sldemo_fuelsys')
```

Use the Simulation Data Inspector programmatic interface to get the signal ID for the signal.

```matlab
runCount = Simulink.sdi.getRunCount;
latestRunID = Simulink.sdi.getRunIDByIndex(runCount);
latestRun = Simulink.sdi.getRun(latestRunID);
speedSigID = latestRun.getSignalIDByIndex(4);
```

Use the signal ID to create a `matlab.io.datastore.sdidatastore` for the speed signal.

```matlab
speedSDIds = matlab.io.datastore.sdidatastore(speedSigID);
```
Verify the Contents of the Datastore

Check the Name property of the `matlab.io.datastore.sdidatastore` to verify that it matches your expectations.

```matlab
speedSDIds.Name
```

ans =
'.map'

You can also use the `preview` method to check that the first ten samples in the signal look correct.

```matlab
speedSDIds.preview
```

ans=10×2 timetable

<table>
<thead>
<tr>
<th>Time</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 sec</td>
<td>0.589</td>
</tr>
<tr>
<td>0.00056199 sec</td>
<td>0.58772</td>
</tr>
<tr>
<td>0.0033719 sec</td>
<td>0.58148</td>
</tr>
<tr>
<td>0.01 sec</td>
<td>0.56765</td>
</tr>
<tr>
<td>0.02 sec</td>
<td>0.54897</td>
</tr>
<tr>
<td>0.03 sec</td>
<td>0.53264</td>
</tr>
<tr>
<td>0.04 sec</td>
<td>0.51837</td>
</tr>
<tr>
<td>0.05 sec</td>
<td>0.50594</td>
</tr>
<tr>
<td>0.055328 sec</td>
<td>0.5</td>
</tr>
<tr>
<td>0.055328 sec</td>
<td>0.5</td>
</tr>
</tbody>
</table>
```

Process Signal Data with the `matlab.io.datastore.sdidatastore`

When your signal is too large to fit into memory, you can use the `readData` method to read chunks of data from the Simulation Data Inspector repository to incrementally process your data. Use the `hasdata` method as the condition for a while loop to incrementally process the whole signal. For example, find the maximum signal value.

```matlab
latestMax = [];  
while speedSDIds.hasdata  
    speedChunk = speedSDIds.read;  
    speedChunkData = speedChunk.Data;  
    latestMax = max([speedChunkData; latestMax]);  
```

```
end

latestMax

latestMax = 0.8897

On each read operation, the `read` method updates the read position for the start of the next read operation. After reading some or all of the `matlab.io.datastore.sdidatastore`, you can reset the read position to start again from the beginning of the signal.

speedSDIds.reset

**Process Signal Data in Memory**

When the signal referenced by your `matlab.io.datastore.sdidatastore` fits into memory, you can use the `readall` method to read all the signal data into memory for processing, rather than reading and processing the data incrementally with the `read` method. The `readall` method returns a `timetable` with all the signal data.

speedTimetable = speedSDIds.readall;

speedMax = max(speedTimetable.Data)

speedMax = 0.8897

**See Also**

Simulink.sdi.DatasetRef | Simulink.sdi.DatasetRef.getAsDatastore | Simulink.sdi.Signal | matlab.io.datastore.SimulationDatastore

**Topics**

“Tall Arrays” (MATLAB)
“Datastore” (MATLAB)

**Introduced in R2017b**
matlab.io.datastore.sdidatastore.hasdata

Class: matlab.io.datastore.sdidatastore  
Package: matlab.io.datastore

Determine if data is available to read

Syntax

tf = sdi_ds.hasdata

Description

tf = sdi_ds.hasdata returns logical 1 if the
matlab.io.datastore.sdidatastore, sdi_ds, has data available to read. When
sdi_ds does not have data available to read, hasdata returns 0.

Output Arguments

tf — Data availability indication
logical

Logical indication of whether the matlab.io.datastore.sdidatastore has data
available to read. When data is available, tf is 1. When data is not available, tf is 0.

Examples

Process Signal Data Using a matlab.io.datastore.sdidatastore

A matlab.io.datastore.sdidatastore references signal data in the Simulation Data
Inspector repository. When the signal is too large to fit into memory, you can use the
matlab.io.datastore.sdidatastore to incrementally process the data manually or
to create a tall timetable for the signal that handles the incremental processing for you.
This example shows how to process data using a `matlab.io.datastore.sdidatastore`.

**Create a `matlab.io.datastore.sdidatastore` for a Signal**

Simulate the `sldemo_fuelsys` model, which is configured to log several signals, to create data in the Simulation Data Inspector repository.

```matlab
sim('sldemo_fuelsys')
```

Use the Simulation Data Inspector programmatic interface to get the signal ID for the signal.

```matlab
runCount = Simulink.sdi.getRunCount;
latestRunID = Simulink.sdi.getRunIDByIndex(runCount);
latestRun = Simulink.sdi.getRun(latestRunID);
speedSigID = latestRun.getSignalIDByIndex(4);
```

Use the signal ID to create a `matlab.io.datastore.sdidatastore` for the speed signal.

```matlab
speedSDIds = matlab.io.datastore.sdidatastore(speedSigID);
```

**Verify the Contents of the Datastore**

Check the Name property of the `matlab.io.datastore.sdidatastore` to verify that it matches your expectations.

```matlab
speedSDIds.Name
```

```
ans =
'map'
```

You can also use the `preview` method to check that the first ten samples in the signal look correct.

```matlab
speedSDIds.preview
```

```
ans=10×2 timetable
     Time    Data
   ______    _____
2.0e+04   1.144
2.0e+04   1.144
2.0e+04   1.144
2.0e+04   1.144
2.0e+04   1.144
2.0e+04   1.144
2.0e+04   1.144
2.0e+04   1.144
2.0e+04   1.144
2.0e+04   1.144
```
Process Signal Data with the `matlab.io.datastore.sdidatastore`

When your signal is too large to fit into memory, you can use the `readData` method to read chunks of data from the Simulation Data Inspector repository to incrementally process your data. Use the `hasdata` method as the condition for a while loop to incrementally process the whole signal. For example, find the maximum signal value.

```matlab
latestMax = []; 

while speedSDIds.hasdata 
    speedChunk = speedSDIds.read; 
    speedChunkData = speedChunk.Data; 
    latestMax = max([speedChunkData; latestMax]);
end

latestMax
```

```matlab
latestMax = 0.8897
```

On each read operation, the `read` method updates the read position for the start of the next read operation. After reading some or all of the `matlab.io.datastore.sdidatastore`, you can reset the read position to start again from the beginning of the signal.

```matlab
speedSDIds.reset
```

Process Signal Data in Memory

When the signal referenced by your `matlab.io.datastore.sdidatastore` fits into memory, you can use the `readall` method to read all the signal data into memory for
processing, rather than reading and processing the data incrementally with the read method. The readall method returns a timetable with all the signal data.

speedTimetable = speedSDIDs.readall;
speedMax = max(speedTimetable.Data)
speedMax = 0.8897

See Also
matlab.io.datastore.sdidatastore

Topics
“Tall Arrays” (MATLAB)
“Datastore” (MATLAB)

Introduced in R2017b
matlab.io.datastore.sdidatastore.preview

Class: matlab.io.datastore.sdidatastore
Package: matlab.io.datastore

Return preview of data in sdidatstore

Syntax

dataPreview = sdi_ds.preview

Description

dataPreview = sdi_ds.preview returns the first 10 samples of signal data in the
matlab.io.datastore.sdidatastore, sdi_ds. The preview method does not
change the read position. Use the preview method to verify that the data in your
matlab.io.datastore.sdidatastore appears as you expect.

Output Arguments

dataPreview — Preview of the data
timetable

First 10 samples of the signal referenced by the
matlab.io.datastore.sdidatastore in a timetable.

Examples

Process Signal Data Using a matlab.io.datastore.sdidatastore

A matlab.io.datastore.sdidatastore references signal data in the Simulation Data
Inspector repository. When the signal is too large to fit into memory, you can use the
matlab.io.datastore.sdidatastore to incrementally process the data manually or
to create a tall timetable for the signal that handles the incremental processing for you.
This example shows how to process data using a `matlab.io.datastore.sdiddatastore`.

**Create a `matlab.io.datastore.sdiddatastore` for a Signal**

Simulate the `sldemo_fuelsys` model, which is configured to log several signals, to create data in the Simulation Data Inspector repository.

```matlab
sim('sldemo_fuelsys')
```

Use the Simulation Data Inspector programmatic interface to get the signal ID for the signal.

```matlab
runCount = Simulink.sdi.getRunCount;
latestRunID = Simulink.sdi.getRunIDByIndex(runCount);
latestRun = Simulink.sdi.getRun(latestRunID);
speedSigID = latestRun.getSignalIDByIndex(4);
```

Use the signal ID to create a `matlab.io.datastore.sdiddatastore` for the speed signal.

```matlab
speedSDIds = matlab.io.datastore.sdiddatastore(speedSigID);
```

**Verify the Contents of the Datastore**

Check the `Name` property of the `matlab.io.datastore.sdiddatastore` to verify that it matches your expectations.

```matlab
speedSDIds.Name
```

```matlab
ans =
 'map'
```

You can also use the `preview` method to check that the first ten samples in the signal look correct.

```matlab
speedSDIds.preview
```

```matlab
ans=10×2 timetable
     Time    Data
      __________    _______
          5        Simulink Classes
          5-16
```
Process Signal Data with the `matlab.io.datastore.sdidatastore`

When your signal is too large to fit into memory, you can use the `readData` method to read chunks of data from the Simulation Data Inspector repository to incrementally process your data. Use the `hasdata` method as the condition for a while loop to incrementally process the whole signal. For example, find the maximum signal value.

```matlab
latestMax = [];  
while speedSDIds.hasdata  
    speedChunk = speedSDIds.read;  
    speedChunkData = speedChunk.Data;  
    latestMax = max([speedChunkData; latestMax]);
end
latestMax
```

`latestMax = 0.8897`

On each read operation, the `read` method updates the read position for the start of the next read operation. After reading some or all of the `matlab.io.datastore.sdidatastore`, you can reset the read position to start again from the beginning of the signal.

```matlab
speedSDIds.reset
```

Process Signal Data in Memory

When the signal referenced by your `matlab.io.datastore.sdidatastore` fits into memory, you can use the `readall` method to read all the signal data into memory for
processing, rather than reading and processing the data incrementally with the `read`
method. The `readall` method returns a `timetable` with all the signal data.

```matlab
speedTimetable = speedSDIds.readall;
speedMax = max(speedTimetable.Data)
speedMax = 0.8897
```

## See Also

- `matlab.io.datastore.sdidatastore`

## Topics

- “Timetables” (MATLAB)

## Introduced in R2017b
matlab.io.datastore.simulationdatastore.sdidatastore.read

Read a chunk of data from an sdidatastore

Syntax

data = sdi_ds.read

Description

data = sdi_ds.read reads a chunk of samples from the matlab.io.datastore.sdidatastore, sdi_ds, and updates the read position for sdi_ds to the point following the endpoint of the returned data. The samples are returned in the timetable, data. The number of samples read by the read method vary, and the returned timetable always fits into memory. Use the read method to incrementally process signals that are too large to fit into memory.

Output Arguments

data — Chunk of data read from sdidatastore

timetable

Chunk of samples read from the matlab.io.datastore.simulationdatastore, returned as a timetable.

Examples

Process Signal Data Using a matlab.io.datastore.sdidatastore

A matlab.io.datastore.sdidatastore references signal data in the Simulation Data Inspector repository. When the signal is too large to fit into memory, you can use the matlab.io.datastore.sdidatastore to incrementally process the data manually or
to create a tall timetable for the signal that handles the incremental processing for you.
This example shows how to process data using a
`matlab.io.datastore.sdidatastore`.

**Create a `matlab.io.datastore.sdidatastore` for a Signal**

Simulate the `sldemo_fuelsys` model, which is configured to log several signals, to
create data in the Simulation Data Inspector repository.

```matlab
sim('sldemo_fuelsys')
```

Use the Simulation Data Inspector programmatic interface to get the signal ID for the
signal.

```matlab
runCount = Simulink.sdi.getRunCount;
latestRunID = Simulink.sdi.getRunIDByIndex(runCount);
latestRun = Simulink.sdi.getRun(latestRunID);
speedSigID = latestRun.getSignalIDByIndex(4);
```

Use the signal ID to create a `matlab.io.datastore.sdidatastore` for the speed
signal.

```matlab
speedSDIds = matlab.io.datastore.sdidatastore(speedSigID);
```

**Verify the Contents of the Datastore**

Check the `Name` property of the `matlab.io.datastore.sdidatastore` to verify that it
matches your expectations.

```matlab
speedSDIds.Name
```

```matlab
ans =
'map'
```

You can also use the `preview` method to check that the first ten samples in the signal
look correct.

```matlab
speedSDIds.preview
```

```matlab
ans=10×2 timetable
     Time    Data
    _______    _______
     5        Simulink Classes
     5-20      5-20
```
Process Signal Data with the `matlab.io.datastore.sdidatastore`

When your signal is too large to fit into memory, you can use the `readData` method to read chunks of data from the Simulation Data Inspector repository to incrementally process your data. Use the `hasdata` method as the condition for a while loop to incrementally process the whole signal. For example, find the maximum signal value.

```matlab
latestMax = [];
while speedSDIds.hasdata
    speedChunk = speedSDIds.read;
    speedChunkData = speedChunk.Data;
    latestMax = max([speedChunkData; latestMax]);
end

latestMax
latestMax = 0.8897
```

On each read operation, the `read` method updates the read position for the start of the next read operation. After reading some or all of the `matlab.io.datastore.sdidatastore`, you can reset the read position to start again from the beginning of the signal.

```matlab
speedSDIds.reset
```

Process Signal Data in Memory

When the signal referenced by your `matlab.io.datastore.sdidatastore` fits into memory, you can use the `readall` method to read all the signal data into memory for
processing, rather than reading and processing the data incrementally with the `read`
method. The `readall` method returns a timetable with all the signal data.

```matlab
speedTimetable = speedSDIds.readall;
speedMax = max(speedTimetable.Data)
speedMax = 0.8897
```

**Alternatives**

You can use your `matlab.io.datastore.sdidatastore` to create a tall timetable to
process signals too large to fit into memory. The tall timetable handles loading and
processing the chunks of signal data for you. The `matlab.io.datastore.sdidatastore` reference page includes an example that shows
how to process your data using a tall timetable. For more information about working with
tall timetables, see “Tall Arrays” (MATLAB).

**See Also**

`matlab.io.datastore.sdidatastore`

**Topics**

“Datastore” (MATLAB)

**Introduced in R2017b**
matlab.io.datastore.sdidatastore.readall

Class: matlab.io.datastore.sdidatastore
Package: matlab.io.datastore

Read all data from an sdidatastore

Syntax

data = sdi_ds.readall

Description

data = sdi_ds.readall reads all the data in the
matlab.io.datastore.sdidatastore, sdi_ds, into memory, returning the
timetable, data. Use readall only when the signal referenced by the
matlab.io.datastore.sdidatastore fits into memory.

Output Arguments

data — timetable of data
timetable

All the data in the matlab.io.datastore.sdidatastore, returned in a timetable.

Examples

Process Signal Data Using a matlab.io.datastore.sdidatastore

A matlab.io.datastore.sdidatastore references signal data in the Simulation Data
Inspector repository. When the signal is too large to fit into memory, you can use the
matlab.io.datastore.sdidatastore to incrementally process the data manually or
to create a tall timetable for the signal that handles the incremental processing for you.
This example shows how to process data using a \texttt{matlab.io.datastore.sdidatastore}.

**Create a \texttt{matlab.io.datastore.sdidatastore} for a Signal**

Simulate the \texttt{sldemo_fuelsys} model, which is configured to log several signals, to create data in the Simulation Data Inspector repository.

\begin{verbatim}
  sim('sldemo_fuelsys')
\end{verbatim}

Use the Simulation Data Inspector programmatic interface to get the signal ID for the signal.

\begin{verbatim}
  runCount = Simulink.sdi.getRunCount;
  latestRunID = Simulink.sdi.getRunIDByIndex(runCount);
  latestRun = Simulink.sdi.getRun(latestRunID);
  speedSigID = latestRun.getSignalIDByIndex(4);
\end{verbatim}

Use the signal ID to create a \texttt{matlab.io.datastore.sdidatastore} for the speed signal.

\begin{verbatim}
  speedSDIds = matlab.io.datastore.sdidatastore(speedSigID);
\end{verbatim}

**Verify the Contents of the Datastore**

Check the \texttt{Name} property of the \texttt{matlab.io.datastore.sdidatastore} to verify that it matches your expectations.

\begin{verbatim}
  speedSDIds.Name
\end{verbatim}

\begin{verbatim}
  ans = 
  'map'
\end{verbatim}

You can also use the \texttt{preview} method to check that the first ten samples in the signal look correct.

\begin{verbatim}
  speedSDIds.preview
\end{verbatim}

\begin{verbatim}
  ans=10×2 timetable
  Time          Data
  ____________    _______
      5           0
      6           0
      7           0
      8           0
      9           0
     10           0
     11           0
     12           0
     13           0
     14           0
\end{verbatim}
Process Signal Data with the `matlab.io.datastore.sdidatastore`

When your signal is too large to fit into memory, you can use the `readData` method to read chunks of data from the Simulation Data Inspector repository to incrementally process your data. Use the `hasdata` method as the condition for a while loop to incrementally process the whole signal. For example, find the maximum signal value.

```matlab
latestMax = [];  
while speedSDIds.hasdata  
    speedChunk = speedSDIds.read;  
    speedChunkData = speedChunk.Data;  
    latestMax = max([speedChunkData; latestMax]);
end

latestMax
```

```
latestMax = 0.8897
```

On each read operation, the `read` method updates the read position for the start of the next read operation. After reading some or all of the `matlab.io.datastore.sdidatastore`, you can reset the read position to start again from the beginning of the signal.

```
speedSDIds.reset
```

**Process Signal Data in Memory**

When the signal referenced by your `matlab.io.datastore.sdidatastore` fits into memory, you can use the `readall` method to read all the signal data into memory for
processing, rather than reading and processing the data incrementally with the read method. The readall method returns a timetable with all the signal data.

```matlab
speedTimetable = speedSDIds.readall;
speedMax = max(speedTimetable.Data)
speedMax = 0.8897
```

**Alternatives**

When your signals fit into memory, you can use other classes and functions of the Simulation Data Inspector programmatic interface, like the Simulink.sdi.Signal class, to access and process simulation data.

**See Also**

- [matlab.io.datastore.sdidatastore](#)

**Topics**

- “Datastore” (MATLAB)

**Introduced in R2017b**
matlab.io.datastore.sdidatastore.reset

Class: matlab.io.datastore.sdidatastore
Package: matlab.io.datastore

Reset the read position

Syntax

sdi_ds.reset

Description

sdi_ds.reset resets the read position for the
matlab.io.datastore.sdidatastore, sdi_ds, to the beginning.

Examples

Process Signal Data Using a matlab.io.datastore.sdidatastore

A matlab.io.datastore.sdidatastore references signal data in the Simulation Data Inspector repository. When the signal is too large to fit into memory, you can use the matlab.io.datastore.sdidatastore to incrementally process the data manually or to create a tall timetable for the signal that handles the incremental processing for you. This example shows how to process data using a matlab.io.datastore.sdidatastore.

Create a matlab.io.datastore.sdidatastore for a Signal

Simulate the sldemo_fuelsys model, which is configured to log several signals, to create data in the Simulation Data Inspector repository.

sim('sldemo_fuelsys')

Use the Simulation Data Inspector programmatic interface to get the signal ID for the signal.
runCount = Simulink.sdi.getRunCount;

latestRunID = Simulink.sdi.getRunIDByIndex(runCount);

latestRun = Simulink.sdi.getRun(latestRunID);

speedSigID = latestRun.getSignalIDByIndex(4);

Use the signal ID to create a matlab.io.datastore.sdidatastore for the speed signal.

speedSDIds = matlab.io.datastore.sdidatastore(speedSigID);

**Verify the Contents of the Datastore**

Check the Name property of the matlab.io.datastore.sdidatastore to verify that it matches your expectations.

speedSDIds.Name

ans =
'map'

You can also use the preview method to check that the first ten samples in the signal look correct.

speedSDIds.preview

ans=10×2 timetable
 Time        Data
    ___        ____
 0 sec       0.589
0.00056199 sec  0.58772
0.0033719 sec   0.58148
0.01 sec       0.56765
0.02 sec       0.54897
0.03 sec       0.53264
0.04 sec       0.51837
0.05 sec       0.50594
0.055328 sec    0.5
0.055328 sec    0.5
Process Signal Data with the `matlab.io.datastore.sdidatastore`

When your signal is too large to fit into memory, you can use the `readData` method to read chunks of data from the Simulation Data Inspector repository to incrementally process your data. Use the `hasdata` method as the condition for a while loop to incrementally process the whole signal. For example, find the maximum signal value.

```matlab
latestMax = [];
while speedSDIds.hasdata
    speedChunk = speedSDIds.read;
    speedChunkData = speedChunk.Data;
    latestMax = max([speedChunkData; latestMax]);
end
latestMax

latestMax = 0.8897
```

On each read operation, the `read` method updates the read position for the start of the next read operation. After reading some or all of the `matlab.io.datastore.sdidatastore`, you can reset the read position to start again from the beginning of the signal.

```matlab
speedSDIds.reset
```

**Process Signal Data in Memory**

When the signal referenced by your `matlab.io.datastore.sdidatastore` fits into memory, you can use the `readall` method to read all the signal data into memory for processing, rather than reading and processing the data incrementally with the `read` method. The `readall` method returns a `timetable` with all the signal data.

```matlab
speedTimetable = speedSDIds.readall;
speedMax = max(speedTimetable.Data)
speedMax = 0.8897
```

**See Also**

`matlab.io.datastore.sdidatastore`
Topics
“Datastore” (MATLAB)

Introduced in R2017b
**matlab.io.datastore.SimulationDatastore class**

**Package:** matlab.io.datastore

Datastore for inputs and outputs of Simulink models

**Description**

A `matlab.io.datastore.SimulationDatastore` object enables a Simulink model to interact with big data. You can load big data as simulation input and log big output data from a simulation. To simulate models with big data, you store the data in a MAT-file and refer to the data through a `SimulationDatastore` object. See “Work with Big Data for Simulations”.

A `SimulationDatastore` object refers to big simulation data (which a MAT-file stores) for one signal. If the MAT-file stores simulation data for a bus signal, a `SimulationDatastore` object refers to the data for one leaf signal element in the bus. You can use the datastore object to inspect and access the data and, through a parent object such as `Simulink.SimulationData.Signal`, simulate a Simulink model with the data.

To analyze the datastore data, you can use the methods and properties of the `SimulationDatastore` object as well as MATLAB tools such as the `tall` function. For more information about the MATLAB tools, see “Getting Started with Datastore” (MATLAB).

**Construction**

After you store big simulation data in a `Simulink.SimulationData.Dataset` object in a MAT-file, a signal element in the `Dataset` object points to the big data. To create a `matlab.io.datastore.SimulationDatastore` object that refers to the big data:

1. At the command prompt or in a script, create a `Simulink.SimulationData.DatasetRef` object that refers to the `Dataset` object in the MAT-file.
2 Use one of these techniques:

- Use one-based, curly-brace indexing (for example, \{1\}) to return an object that represents the target signal element, such as `Simulink.SimulationData.Signal` or `Simulink.SimulationData.State`. For example, for a `DatasetRef` object named `logsout_ref`, to create a `Signal` object that refers to the second signal element, use this code:

  ```
  myLoggedSig = logsout_ref\{2\}
  ```

- Use the `getAsDatastore` method of the `DatasetRef` object to return an object that represents the target signal element. For more information, see `Simulink.SimulationData.DatasetRef.getAsDatastore`.

The `SimulationDatastore` object resides in the `Values` property of the returned object.

**Properties**

**FileName — Name and path of file that contains big data**
character vector

Name and path of the file that contains the big data, returned as a character vector. This property is read-only.

Data Types: char

**NumSamples — Total number of samples (time steps) in the datastore**
integer

Total number of samples (time steps) in the datastore, returned as an integer. The `readall` method extracts this many samples from the big data. This property is read-only.

Data Types: `uint64`

**ReadSize — Amount of data to read at a time**
100 (default) | scalar double

Amount of data to read at a time, in number of samples (time steps), specified as a scalar double. The `read` method extracts this many samples from the big data.

Data Types: double
Methods

hasdata  Determine if data is available to read
preview  Return subset of data from datastore
progress Return percentage of data that you have read from a datastore
read     Read data in datastore
readall  Read all data in datastore
reset    Reset datastore to initial state

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects (MATLAB).

Limitations

- SimulationDatastore does not support using a parallel pool with Parallel Computing Toolbox installed. To analyze data using tall arrays or run MapReduce algorithms, set the global execution environment to be the local MATLAB session using mapreducer. Enter this code:

  mapreducer(0)

  For information about controlling parallel resources, see “Run mapreduce on a Parallel Pool” (Parallel Computing Toolbox).

- You cannot use a MATLAB tall variable as simulation input data.

Examples

Inspect and Analyze Data in Simulation Datastore

This example shows how to log big data from a simulation and inspect and analyze portions of that data by interacting with a matlab.io.datastore.SimulationDatastore object.
Log Big Data from Model

Open the example model sldemo_fuelsys.

```
open_system('sldemo_fuelsys')
```

Select Configuration Parameters > Data Import/Export > Log Dataset data to file.

```
set_param('sldemo_fuelsys','LoggingToFile','on')
```

Simulate the model.

```
sim('sldemo_fuelsys')
```

The MAT-file `out.mat` appears in your current folder. The file contains data for logged signals such as `fuel` (which is at the root level of the model).

At the command prompt, create a `DatasetRef` object that refers to the logging variable by name, `sldemo_fuelsys_output`. 
DSRef = Simulink.SimulationData.DatasetRef('out.mat','sldemo_fuelsys_output');

**Preview Big Data**

Use curly braces ({ and }) to extract the signal element fuel, which is the tenth element in DSRef, as a Simulink.SimulationData.Signal object that contains a SimulationDatastore object.

SimDataSig = DSRef{10};

To more easily interact with the SimulationDatastore object that resides in the Values property of the Signal object, store a handle in a variable named DStore.

DStore = SimDataSig.Values;

Use the preview method to inspect the first five samples of logged data for the fuel signal.

preview(DStore)

ans =

10x1 timetable

<table>
<thead>
<tr>
<th>Time</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 sec</td>
<td>1.209</td>
</tr>
<tr>
<td>0.00056199 sec</td>
<td>1.209</td>
</tr>
<tr>
<td>0.0033719 sec</td>
<td>1.209</td>
</tr>
<tr>
<td>0.01 sec</td>
<td>1.1729</td>
</tr>
<tr>
<td>0.02 sec</td>
<td>1.1409</td>
</tr>
<tr>
<td>0.03 sec</td>
<td>1.1124</td>
</tr>
<tr>
<td>0.04 sec</td>
<td>1.0873</td>
</tr>
<tr>
<td>0.05 sec</td>
<td>1.0652</td>
</tr>
<tr>
<td>0.055328 sec</td>
<td>1.0652</td>
</tr>
</tbody>
</table>

**Inspect Specific Sample**

Inspect the 603rd sample of logged fuel data.
Set the **ReadSize** property of **DStore** to a number that, considering memory resources, your computer can tolerate. For example, set **ReadSize** to 200.

```matlab
DStore.ReadSize = 200;
```

Read from the datastore three times. Each read operation advances the reading position by 200 samples.

```matlab
read(DStore);
read(DStore);
read(DStore);
```

Now that you are very close to the 603rd sample, set **ReadSize** to a smaller number. For example, set **ReadSize** to 5.

```matlab
DStore.ReadSize = 5;
```

Read from the datastore again.

```matlab
read(DStore)
```

```
ans =

5x1 timetable

<table>
<thead>
<tr>
<th>Time</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.79 sec</td>
<td>1.6097</td>
</tr>
<tr>
<td>5.8 sec</td>
<td>1.6136</td>
</tr>
<tr>
<td>5.81 sec</td>
<td>1.6003</td>
</tr>
<tr>
<td>5.82 sec</td>
<td>1.5904</td>
</tr>
<tr>
<td>5.83 sec</td>
<td>1.5832</td>
</tr>
</tbody>
</table>
```

The third sample of read data is the 603rd sample in the datastore.

**Inspect Earlier Sample**

Inspect the 403rd sample of logged fuel data. Due to previous read operations, the datastore now reads starting from the 606th sample, so you must reset the datastore. Then, you can read from the first sample up to the 403rd sample.

Use the `reset` method to reset **DStore**.
reset(DStore);

Set ReadSize to 200 again.

DStore.ReadSize = 200;

Read from the datastore twice to advance the read position to the 401st sample.

read(DStore);
read(DStore);

Set ReadSize to 5 again.

DStore.ReadSize = 5;

Read from the datastore.

read(DStore)

ans =

5x1 timetable

<table>
<thead>
<tr>
<th>Time</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.85 sec</td>
<td>0.999</td>
</tr>
<tr>
<td>3.86 sec</td>
<td>0.99219</td>
</tr>
<tr>
<td>3.87 sec</td>
<td>0.98538</td>
</tr>
<tr>
<td>3.88 sec</td>
<td>0.97858</td>
</tr>
<tr>
<td>3.89 sec</td>
<td>0.97179</td>
</tr>
</tbody>
</table>

**Extract Multiple Samples**

Extract samples 1001 through 1020 (a total of 20 samples).

Reset the datastore.

reset(DStore)

Advance to sample 1001.

DStore.ReadSize = 200;
for i = 1:5
    read(DStore);
end

Prepare to extract 20 samples from the datastore.

DStore.ReadSize = 20;

Extract samples 1001 through 1020. Store the extracted data in a variable named targetSamples.

targetSamples = read(DStore)

targetSamples =

20x1 timetable

<table>
<thead>
<tr>
<th>Time</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.7 sec</td>
<td>1.5828</td>
</tr>
<tr>
<td>9.71 sec</td>
<td>1.5733</td>
</tr>
<tr>
<td>9.72 sec</td>
<td>1.5664</td>
</tr>
<tr>
<td>9.73 sec</td>
<td>1.5614</td>
</tr>
<tr>
<td>9.74 sec</td>
<td>1.5579</td>
</tr>
<tr>
<td>9.75 sec</td>
<td>1.5553</td>
</tr>
<tr>
<td>9.76 sec</td>
<td>1.5703</td>
</tr>
<tr>
<td>9.77 sec</td>
<td>1.582</td>
</tr>
<tr>
<td>9.78 sec</td>
<td>1.5913</td>
</tr>
<tr>
<td>9.79 sec</td>
<td>1.5988</td>
</tr>
<tr>
<td>9.8 sec</td>
<td>1.605</td>
</tr>
<tr>
<td>9.81 sec</td>
<td>1.6101</td>
</tr>
<tr>
<td>9.82 sec</td>
<td>1.6145</td>
</tr>
<tr>
<td>9.83 sec</td>
<td>1.6184</td>
</tr>
<tr>
<td>9.84 sec</td>
<td>1.6049</td>
</tr>
<tr>
<td>9.85 sec</td>
<td>1.595</td>
</tr>
<tr>
<td>9.86 sec</td>
<td>1.5877</td>
</tr>
<tr>
<td>9.87 sec</td>
<td>1.5824</td>
</tr>
<tr>
<td>9.88 sec</td>
<td>1.5785</td>
</tr>
<tr>
<td>9.89 sec</td>
<td>1.5757</td>
</tr>
</tbody>
</table>

Find Maximum Value of Data in Datastore

Reset the datastore.
reset(DStore)

Write a while loop, using the hasdata method, to incrementally analyze the data in chunks of 200 samples.

DStore.ReadSize = 200;
runningMax = [];
while hasdata(DStore)
    tt = read(DStore);
    rawChunk = tt.Data;
    runningMax = max([rawChunk; runningMax]);
end

Now, the variable runningMax stores the maximum value in the entire datastore.

runningMax

runningMax =

1.6423

See Also

Topics
“Work with Big Data for Simulations”

Introduced in R2017a
hasdata

Class: matlab.io.datastore.SimulationDatastore
Package: matlab.io.datastore

Determine if data is available to read

Syntax

\[ \text{tf} = \text{hasdata}(\text{dst}) \]

Description

\[ \text{tf} = \text{hasdata}(\text{dst}) \] returns logical 1 (true) if there is data available to read from the datastore (matlab.io.datastore.SimulationDatastore object) specified by \text{dst}. Otherwise, it returns logical 0 (false).

Input Arguments

dst — Input datastore
matlab.io.datastore.SimulationDatastore object

Input datastore, specified as a matlab.io.datastore.SimulationDatastore object. To create a SimulationDatastore object, see matlab.io.datastore.SimulationDatastore.

Examples

Inspect and Analyze Data in Simulation Datastore

This example shows how to log big data from a simulation and inspect and analyze portions of that data by interacting with a matlab.io.datastore.SimulationDatastore object.
Log Big Data from Model

Open the example model sldemo_fuelsys.

open_system('sldemo_fuelsys')

Fault-Tolerant Fuel Control System

Select Configuration Parameters > Data Import/Export > Log Dataset data to file.

set_param('sldemo_fuelsys','LoggingToFile','on')

Simulate the model.

sim('sldemo_fuelsys')

The MAT-file out.mat appears in your current folder. The file contains data for logged signals such as fuel (which is at the root level of the model).

At the command prompt, create a DatasetRef object that refers to the logging variable by name, sldemo_fuelsys_output.
DSRef = Simulink.SimulationData.DatasetRef('out.mat','sldemo_fuelsys_output');

**Preview Big Data**

Use curly braces ({ and }) to extract the signal element `fuel`, which is the tenth element in DSRef, as a Simulink.SimulationData.Signal object that contains a SimulationDatastore object.

SimDataSig = DSRef{10};

To more easily interact with the SimulationDatastore object that resides in the Values property of the Signal object, store a handle in a variable named DStore.

DStore = SimDataSig.Values;

Use the `preview` method to inspect the first five samples of logged data for the `fuel` signal.

preview(DStore)

```
ans =

10x1 timetable

     Time        Data
    ______    ______
    0 sec     1.209
 0.00056199 sec   1.209
 0.0033719 sec   1.209
     0.01 sec   1.1729
     0.02 sec   1.1409
     0.03 sec   1.1124
     0.04 sec   1.0873
     0.05 sec   1.0652
0.055328 sec     1.0652
0.055328 sec     1.0652
```

**Inspect Specific Sample**

Inspect the 603rd sample of logged fuel data.
Set the ReadSize property of DStore to a number that, considering memory resources, your computer can tolerate. For example, set ReadSize to 200.

DStore.ReadSize = 200;

Read from the datastore three times. Each read operation advances the reading position by 200 samples.

read(DStore);
read(DStore);
read(DStore);

Now that you are very close to the 603rd sample, set ReadSize to a smaller number. For example, set ReadSize to 5.

DStore.ReadSize = 5;

Read from the datastore again.

read(DStore)

ans =

5x1 timetable

<table>
<thead>
<tr>
<th>Time</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.79 sec</td>
<td>1.6097</td>
</tr>
<tr>
<td>5.8 sec</td>
<td>1.6136</td>
</tr>
<tr>
<td>5.81 sec</td>
<td>1.6003</td>
</tr>
<tr>
<td>5.82 sec</td>
<td>1.5904</td>
</tr>
<tr>
<td>5.83 sec</td>
<td>1.5832</td>
</tr>
</tbody>
</table>

The third sample of read data is the 603rd sample in the datastore.

**Inspect Earlier Sample**

Inspect the 403rd sample of logged fuel data. Due to previous read operations, the datastore now reads starting from the 606th sample, so you must reset the datastore. Then, you can read from the first sample up to the 403rd sample.

Use the reset method to reset DStore.
reset(DStore);
Set ReadSize to 200 again.
DStore.ReadSize = 200;
Read from the datastore twice to advance the read position to the 401st sample.
read(DStore);
read(DStore);
Set ReadSize to 5 again.
DStore.ReadSize = 5;
Read from the datastore.
read(DStore)

ans =

5x1 timetable

<table>
<thead>
<tr>
<th>Time</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.85 sec</td>
<td>0.999</td>
</tr>
<tr>
<td>3.86 sec</td>
<td>0.99219</td>
</tr>
<tr>
<td>3.87 sec</td>
<td>0.98538</td>
</tr>
<tr>
<td>3.88 sec</td>
<td>0.97858</td>
</tr>
<tr>
<td>3.89 sec</td>
<td>0.97179</td>
</tr>
</tbody>
</table>

**Extract Multiple Samples**

Extract samples 1001 through 1020 (a total of 20 samples).

Reset the datastore.
reset(DStore)
Advance to sample 1001.
DStore.ReadSize = 200;
for i = 1:5
    read(DStore);
end

Prepare to extract 20 samples from the datastore.

DStore.ReadSize = 20;

Extract samples 1001 through 1020. Store the extracted data in a variable named targetSamples.

targetSamples = read(DStore)

targetSamples =

20x1 timetable

<table>
<thead>
<tr>
<th>Time</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.7 sec</td>
<td>1.5828</td>
</tr>
<tr>
<td>9.71 sec</td>
<td>1.5733</td>
</tr>
<tr>
<td>9.72 sec</td>
<td>1.5664</td>
</tr>
<tr>
<td>9.73 sec</td>
<td>1.5614</td>
</tr>
<tr>
<td>9.74 sec</td>
<td>1.5579</td>
</tr>
<tr>
<td>9.75 sec</td>
<td>1.5553</td>
</tr>
<tr>
<td>9.76 sec</td>
<td>1.5703</td>
</tr>
<tr>
<td>9.77 sec</td>
<td>1.582</td>
</tr>
<tr>
<td>9.78 sec</td>
<td>1.5913</td>
</tr>
<tr>
<td>9.79 sec</td>
<td>1.5988</td>
</tr>
<tr>
<td>9.8 sec</td>
<td>1.605</td>
</tr>
<tr>
<td>9.81 sec</td>
<td>1.6101</td>
</tr>
<tr>
<td>9.82 sec</td>
<td>1.6145</td>
</tr>
<tr>
<td>9.83 sec</td>
<td>1.6184</td>
</tr>
<tr>
<td>9.84 sec</td>
<td>1.6049</td>
</tr>
<tr>
<td>9.85 sec</td>
<td>1.595</td>
</tr>
<tr>
<td>9.86 sec</td>
<td>1.5877</td>
</tr>
<tr>
<td>9.87 sec</td>
<td>1.5824</td>
</tr>
<tr>
<td>9.88 sec</td>
<td>1.5785</td>
</tr>
<tr>
<td>9.89 sec</td>
<td>1.5757</td>
</tr>
</tbody>
</table>

Find Maximum Value of Data in Datastore

Reset the datastore.
reset(DStore)

Write a while loop, using the hasdata method, to incrementally analyze the data in chunks of 200 samples.

DStore.ReadSize = 200;
runningMax = [];
while hasdata(DStore)
    tt = read(DStore);
    rawChunk = tt.Data;
    runningMax = max([rawChunk; runningMax]);
end

Now, the variable runningMax stores the maximum value in the entire datastore.

runningMax

runningMax =

1.6423

See Also

Topics
“Work with Big Data for Simulations”

Introduced in R2017a
preview

Class: matlab.io.datastore.SimulationDatastore
Package: matlab.io.datastore

Return subset of data from datastore

Syntax

data = preview(dst)

Description

data = preview(dst) returns a subset of data from datastore (matlab.io.datastore.SimulationDatastore object) dst without changing its current read position. preview returns only the first ten samples (time steps) of data in the datastore. Use this method to quickly inspect and verify that the data appears as you expect.

Input Arguments

dst — Input datastore
matlab.io.datastore.SimulationDatastore object

Input datastore, specified as a matlab.io.datastore.SimulationDatastore object. To create a SimulationDatastore object, see matlab.io.datastore.SimulationDatastore.

Output Arguments

data — Subset of data
timetable object
Subset of data, returned as a timetable object. For information about timetable, see “Timetables” (MATLAB).

Examples

Inspect and Analyze Data in Simulation Datastore

This example shows how to log big data from a simulation and inspect and analyze portions of that data by interacting with a matlab.io.datastore.SimulationDatastore object.

Log Big Data from Model

Open the example model sldemo_fuelsys.

open_system('sldemo_fuelsys')
Select **Configuration Parameters > Data Import/Export > Log Dataset data to file.**

```matlab
set_param('sldemo_fuelsys','LoggingToFile','on')
```

Simulate the model.

```matlab
sim('sldemo_fuelsys')
```

The MAT-file `out.mat` appears in your current folder. The file contains data for logged signals such as `fuel` (which is at the root level of the model).

At the command prompt, create a `DatasetRef` object that refers to the logging variable by name, `sldemo_fuelsys_output`.

```matlab
DSRef = Simulink.SimulationData.DatasetRef('out.mat','sldemo_fuelsys_output');
```

**Preview Big Data**

Use curly braces (`{` and `}`) to extract the signal element `fuel`, which is the tenth element in `DSRef`, as a `Simulink.SimulationData.Signal` object that contains a `SimulationDatastore` object.

```matlab
SimDataSig = DSRef{10};
```

To more easily interact with the `SimulationDatastore` object that resides in the `Values` property of the `Signal` object, store a handle in a variable named `DStore`.

```matlab
DStore = SimDataSig.Values;
```

Use the `preview` method to inspect the first five samples of logged data for the `fuel` signal.

```matlab
preview(DStore)
```

```
ans =

10x1 timetable

<table>
<thead>
<tr>
<th>Time</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 sec</td>
<td>1.209</td>
</tr>
<tr>
<td>0.00056199 sec</td>
<td>1.209</td>
</tr>
<tr>
<td>0.0033719 sec</td>
<td>1.209</td>
</tr>
</tbody>
</table>
```

5-49
Inspect Specific Sample

Inspect the 603rd sample of logged fuel data.

Set the ReadSize property of DStore to a number that, considering memory resources, your computer can tolerate. For example, set ReadSize to 200.

DStore.ReadSize = 200;

Read from the datastore three times. Each read operation advances the reading position by 200 samples.

read(DStore);
read(DStore);
read(DStore);

Now that you are very close to the 603rd sample, set ReadSize to a smaller number. For example, set ReadSize to 5.

DStore.ReadSize = 5;

Read from the datastore again.

read(DStore)

ans =

5x1 timetable

<table>
<thead>
<tr>
<th>Time</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.79 sec</td>
<td>1.6097</td>
</tr>
<tr>
<td>5.8 sec</td>
<td>1.6136</td>
</tr>
<tr>
<td>5.81 sec</td>
<td>1.6003</td>
</tr>
</tbody>
</table>
The third sample of read data is the 603rd sample in the datastore.

**Inspect Earlier Sample**

Inspect the 403rd sample of logged fuel data. Due to previous read operations, the datastore now reads starting from the 606th sample, so you must reset the datastore. Then, you can read from the first sample up to the 403rd sample.

Use the reset method to reset DStore.

```matlab
reset(DStore);
```

Set ReadSize to 200 again.

```matlab
DStore.ReadSize = 200;
```

Read from the datastore twice to advance the read position to the 401st sample.

```matlab
read(DStore);
read(DStore);
```

Set ReadSize to 5 again.

```matlab
DStore.ReadSize = 5;
```

Read from the datastore.

```matlab
read(DStore)
```

```matlab
ans =

5x1 timetable

<table>
<thead>
<tr>
<th>Time</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.85 sec</td>
<td>0.999</td>
</tr>
<tr>
<td>3.86 sec</td>
<td>0.99219</td>
</tr>
<tr>
<td>3.87 sec</td>
<td>0.98538</td>
</tr>
<tr>
<td>3.88 sec</td>
<td>0.97858</td>
</tr>
</tbody>
</table>
```
Extract Multiple Samples

Extract samples 1001 through 1020 (a total of 20 samples).

Reset the datastore.
reset(DStore)

Advance to sample 1001.
DStore.ReadSize = 200;

for i = 1:5
    read(DStore);
end

Prepare to extract 20 samples from the datastore.
DStore.ReadSize = 20;

Extract samples 1001 through 1020. Store the extracted data in a variable named targetSamples.
targetSamples = read(DStore)

targetSamples =

20x1 timetable

<table>
<thead>
<tr>
<th>Time</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.7 sec</td>
<td>1.5828</td>
</tr>
<tr>
<td>9.71 sec</td>
<td>1.5733</td>
</tr>
<tr>
<td>9.72 sec</td>
<td>1.5664</td>
</tr>
<tr>
<td>9.73 sec</td>
<td>1.5614</td>
</tr>
<tr>
<td>9.74 sec</td>
<td>1.5579</td>
</tr>
<tr>
<td>9.75 sec</td>
<td>1.5553</td>
</tr>
<tr>
<td>9.76 sec</td>
<td>1.5703</td>
</tr>
<tr>
<td>9.77 sec</td>
<td>1.582</td>
</tr>
<tr>
<td>9.78 sec</td>
<td>1.5913</td>
</tr>
</tbody>
</table>
Find Maximum Value of Data in Datastore

Reset the datastore.

```matlab
reset(DStore)
```

Write a while loop, using the hasdata method, to incrementally analyze the data in chunks of 200 samples.

```matlab
DStore.ReadSize = 200;
runningMax = [];
while hasdata(DStore)
    tt = read(DStore);
    rawChunk = tt.Data;
    runningMax = max([rawChunk; runningMax]);
end
```

Now, the variable `runningMax` stores the maximum value in the entire datastore.

```matlab
runningMax = 
```
See Also

Topics
“Work with Big Data for Simulations”

Introduced in R2017a
**progress**

*Class:* `matlab.io.datastore.SimulationDatastore`  
*Package:* `matlab.io.datastore`  

Return percentage of data that you have read from a datastore

**Syntax**

\[ p = \text{progress}(\text{dst}) \]

**Description**

\[ p = \text{progress}(\text{dst}) \] returns the percentage, as a number between 0 and 1, of the data that you have read from a datastore (`matlab.io.datastore.SimulationDatastore` object). For example, a return value of 0.55 means you have read 55% of the data. Use the `progress` method and the `NumSamples` property to determine the current read position.

You read data from a `SimulationDatastore` object by using the `read` method.

**Input Arguments**

\[ \text{dst} \] — Input datastore  
`matlab.io.datastore.SimulationDatastore` object  

Input datastore, specified as a `matlab.io.datastore.SimulationDatastore` object. To create a `SimulationDatastore` object, see `matlab.io.datastore.SimulationDatastore`.

**Output Arguments**

\[ p \] — Percentage of data that you have read from the datastore  
scalar double
Percentage of data that you have read from the datastore, returned as a scalar double.

Data Types: double

See Also

Topics
“Work with Big Data for Simulations”

Introduced in R2017a
read

Class: matlab.io.datastore.SimulationDatastore
Package: matlab.io.datastore

Read data in datastore

Syntax

data = read(dst)
[data,info] = read(dst)

Description

data = read(dst) returns data from a datastore (matlab.io.datastore.SimulationDatastore object). Subsequent calls to the read function continue reading from the endpoint of the previous call. Use the ReadSize property of the SimulationDatastore object to specify the amount of data, in samples (time steps), to read at a time. Use the progress method and the NumSamples property to determine the current read position.

[data,info] = read(dst) also returns information about the extracted data in info.

Input Arguments

dst — Input datastore
matlab.io.datastore.SimulationDatastore object

Input datastore, specified as a matlab.io.datastore.SimulationDatastore object. To create a SimulationDatastore object, see matlab.io.datastore.SimulationDatastore.
Output Arguments

data — Output data
timetable object

Output data, returned as a timetable object. For information about timetable, see “Timetables” (MATLAB).

info — Information about read data
structure array

Information about read data, returned as a structure. The structure has one field, FileName, which is a fully resolved path containing the path string, the name of the file, and the file extension.

Examples

Inspect and Analyze Data in Simulation Datastore

This example shows how to log big data from a simulation and inspect and analyze portions of that data by interacting with a matlab.io.datastore.SimulationDatastore object.

Log Big Data from Model

Open the example model sldemo_fuelsys.

open_system('sldemo_fuelsys')
Select **Configuration Parameters > Data Import/Export > Log Dataset data to file.**

```matlab
set_param('sldemo_fuelsys','LoggingToFile','on')
```

Simulate the model.

```matlab
sim('sldemo_fuelsys')
```

The MAT-file `out.mat` appears in your current folder. The file contains data for logged signals such as fuel (which is at the root level of the model).

At the command prompt, create a DatasetRef object that refers to the logging variable by name, `sldemo_fuelsys_output`.

```matlab
DSRef = Simulink.SimulationData.DatasetRef('out.mat','sldemo_fuelsys_output');
```
Preview Big Data

Use curly braces (\{ and \}) to extract the signal element fuel, which is the tenth element in DSRef, as a Simulink.SimulationData.Signal object that contains a SimulationDatastore object.

```
SimDataSig = DSRef{10};
```

To more easily interact with the SimulationDatastore object that resides in the Values property of the Signal object, store a handle in a variable named DStore.

```
DStore = SimDataSig.Values;
```

Use the preview method to inspect the first five samples of logged data for the fuel signal.

```
preview(DStore)
```

```
ans =

10x1 timetable

         Time    Data
    __________    ______
          0 sec    1.209
     0.00056199 sec    1.209
     0.0033719 sec    1.209
     0.01 sec    1.1729
     0.02 sec    1.1409
     0.03 sec    1.1124
     0.04 sec    1.0873
     0.05 sec    1.0652
     0.055328 sec    1.0652
     0.055328 sec    1.0652
```

Inspect Specific Sample

Inspect the 603rd sample of logged fuel data.

Set the ReadSize property of DStore to a number that, considering memory resources, your computer can tolerate. For example, set ReadSize to 200.
DStore.ReadSize = 200;

Read from the datastore three times. Each read operation advances the reading position by 200 samples.

read(DStore);
read(DStore);
read(DStore);

Now that you are very close to the 603rd sample, set ReadSize to a smaller number. For example, set ReadSize to 5.

DStore.ReadSize = 5;

Read from the datastore again.

read(DStore)

ans =

5x1 timetable

<table>
<thead>
<tr>
<th>Time</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.79 sec</td>
<td>1.6097</td>
</tr>
<tr>
<td>5.8 sec</td>
<td>1.6136</td>
</tr>
<tr>
<td>5.81 sec</td>
<td>1.6003</td>
</tr>
<tr>
<td>5.82 sec</td>
<td>1.5904</td>
</tr>
<tr>
<td>5.83 sec</td>
<td>1.5832</td>
</tr>
</tbody>
</table>

The third sample of read data is the 603rd sample in the datastore.

**Inspect Earlier Sample**

Inspect the 403rd sample of logged fuel data. Due to previous read operations, the datastore now reads starting from the 606th sample, so you must reset the datastore. Then, you can read from the first sample up to the 403rd sample.

Use the reset method to reset DStore.

reset(DStore);
Set `ReadSize` to 200 again.

```matlab
DStore.ReadSize = 200;
```

Read from the datastore twice to advance the read position to the 401st sample.

```matlab
read(DStore);
read(DStore);
```

Set `ReadSize` to 5 again.

```matlab
DStore.ReadSize = 5;
```

Read from the datastore.

```matlab
read(DStore)
```

```matlab
ans =

5x1 timetable

<table>
<thead>
<tr>
<th>Time</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.85 sec</td>
<td>0.999</td>
</tr>
<tr>
<td>3.86 sec</td>
<td>0.99219</td>
</tr>
<tr>
<td>3.87 sec</td>
<td>0.98538</td>
</tr>
<tr>
<td>3.88 sec</td>
<td>0.97858</td>
</tr>
<tr>
<td>3.89 sec</td>
<td>0.97179</td>
</tr>
</tbody>
</table>
```

**Extract Multiple Samples**

Extract samples 1001 through 1020 (a total of 20 samples).

Reset the datastore.

```matlab
reset(DStore)
```

Advance to sample 1001.

```matlab
DStore.ReadSize = 200;
```

```matlab
for i = 1:5
```
read(DStore);
end

Prepare to extract 20 samples from the datastore.

DStore.ReadSize = 20;

Extract samples 1001 through 1020. Store the extracted data in a variable named targetSamples.

targetSamples = read(DStore)

targetSamples =

20x1 timetable

<table>
<thead>
<tr>
<th>Time</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.7 sec</td>
<td>1.5828</td>
</tr>
<tr>
<td>9.71 sec</td>
<td>1.5733</td>
</tr>
<tr>
<td>9.72 sec</td>
<td>1.5664</td>
</tr>
<tr>
<td>9.73 sec</td>
<td>1.5614</td>
</tr>
<tr>
<td>9.74 sec</td>
<td>1.5579</td>
</tr>
<tr>
<td>9.75 sec</td>
<td>1.5553</td>
</tr>
<tr>
<td>9.76 sec</td>
<td>1.5703</td>
</tr>
<tr>
<td>9.77 sec</td>
<td>1.582</td>
</tr>
<tr>
<td>9.78 sec</td>
<td>1.5913</td>
</tr>
<tr>
<td>9.79 sec</td>
<td>1.5988</td>
</tr>
<tr>
<td>9.8 sec</td>
<td>1.605</td>
</tr>
<tr>
<td>9.81 sec</td>
<td>1.6101</td>
</tr>
<tr>
<td>9.82 sec</td>
<td>1.6145</td>
</tr>
<tr>
<td>9.83 sec</td>
<td>1.6184</td>
</tr>
<tr>
<td>9.84 sec</td>
<td>1.6049</td>
</tr>
<tr>
<td>9.85 sec</td>
<td>1.595</td>
</tr>
<tr>
<td>9.86 sec</td>
<td>1.5877</td>
</tr>
<tr>
<td>9.87 sec</td>
<td>1.5824</td>
</tr>
<tr>
<td>9.88 sec</td>
<td>1.5785</td>
</tr>
<tr>
<td>9.89 sec</td>
<td>1.5757</td>
</tr>
</tbody>
</table>

Find Maximum Value of Data in Datstore

Reset the datastore.
reset(DStore)

Write a while loop, using the hasdata method, to incrementally analyze the data in chunks of 200 samples.

DStore.ReadSize = 200;
runningMax = [];
while hasdata(DStore)    
    tt = read(DStore);
    rawChunk = tt.Data;
    runningMax = max([rawChunk; runningMax]);
end

Now, the variable runningMax stores the maximum value in the entire datastore.

runningMax

    runningMax =

    1.6423

See Also

Topics
“Work with Big Data for Simulations”

Introduced in R2017a
readall

Class: matlab.io.datastore.SimulationDatastore
Package: matlab.io.datastore

Read all data in datastore

Syntax

data = readall(dst)

Description

data = readall(dst) returns all the data in the datastore (matlab.io.datastore.SimulationDatastore object) specified by dst.

If all the data in the datastore does not fit in memory, readall returns an error. To determine how many samples (time steps) a datastore holds, inspect the NumSamples property of the SimulationDatastore object.

Input Arguments

dst — Input datastore
matlab.io.datastore.SimulationDatastore object

Input datastore, specified as a matlab.io.datastore.SimulationDatastore object. To create a SimulationDatastore object, see matlab.io.datastore.SimulationDatastore.

Output Arguments

data — All data in the datastore
timetable object
All data in the datastore, returned as a `timetable` object. For information about `timetable`, see “Timetables” (MATLAB).

### See Also

#### Topics

“Work with Big Data for Simulations”

**Introduced in R2017a**
reset

Class: matlab.io.datastore.SimulationDatastore
Package: matlab.io.datastore

Reset datastore to initial state

Syntax

reset(dst)

Description

reset(dst) sets the read position of the datastore (matlab.io.datastore.SimulationDatastore object) specified by dst to the first sample in the datastore. Use reset to reread data from a datastore. You read from a datastore by using the read method.

Input Arguments

dst — Input datastore
matlab.io.datastore.SimulationDatastore object

Input datastore, specified as a matlab.io.datastore.SimulationDatastore object. To create a SimulationDatastore object, see matlab.io.datastore.SimulationDatastore.

Examples
Inspect and Analyze Data in Simulation Datastore

This example shows how to log big data from a simulation and inspect and analyze portions of that data by interacting with a `matlab.io.datastore.SimulationDatastore` object.

Log Big Data from Model

Open the example model `sldemo_fuelsys`.

```matlab
open_system('sldemo_fuelsys')
```

Select `Configuration Parameters > Data Import/Export > Log Dataset data to file`.

```matlab
set_param('sldemo_fuelsys','LoggingToFile','on')
```

Simulate the model.

```matlab
sim('sldemo_fuelsys')
```
The MAT-file `out.mat` appears in your current folder. The file contains data for logged signals such as `fuel` (which is at the root level of the model).

At the command prompt, create a `DatasetRef` object that refers to the logging variable by name, `sldemo_fuelsys_output`.

```
DSRef = Simulink.SimulationData.DatasetRef('out.mat','sldemo_fuelsys_output');
```

**Preview Big Data**

Use curly braces (`{` and `}`) to extract the signal element `fuel`, which is the tenth element in `DSRef`, as a `Simulink.SimulationData.Signal` object that contains a `SimulationDatastore` object.

```
SimDataSig = DSRef{10};
```

To more easily interact with the `SimulationDatastore` object that resides in the `Values` property of the `Signal` object, store a handle in a variable named `DStore`.

```
DStore = SimDataSig.Values;
```

Use the `preview` method to inspect the first five samples of logged data for the `fuel` signal.

```
preview(DStore)
```

```
ans =

10x1 timetable

         Time    Data
        _______    ______
    0 sec              1.209
0.00056199 sec     1.209
0.0033719 sec      1.209
0.01 sec          1.1729
0.02 sec          1.1409
0.03 sec          1.1124
0.04 sec          1.0873
0.05 sec          1.0652
0.055328 sec      1.0652
0.055328 sec      1.0652
```
Inspect Specific Sample

Inspect the 603rd sample of logged fuel data.

Set the ReadSize property of DStore to a number that, considering memory resources, your computer can tolerate. For example, set ReadSize to 200.

DStore.ReadSize = 200;

Read from the datastore three times. Each read operation advances the reading position by 200 samples.

read(DStore);
read(DStore);
read(DStore);

Now that you are very close to the 603rd sample, set ReadSize to a smaller number. For example, set ReadSize to 5.

DStore.ReadSize = 5;

Read from the datastore again.

read(DStore)

ans =
    5x1 timetable
      Time    Data
    __________    ______
       5.79 sec  1.6097
       5.8  sec  1.6136
       5.81 sec  1.6003
       5.82 sec  1.5904
       5.83 sec  1.5832

The third sample of read data is the 603rd sample in the datastore.
Inspect Earlier Sample

Inspect the 403rd sample of logged fuel data. Due to previous read operations, the datastore now reads starting from the 606th sample, so you must reset the datastore. Then, you can read from the first sample up to the 403rd sample.

Use the reset method to reset DStore.

```matlab
reset(DStore);
```

Set ReadSize to 200 again.

```matlab
DStore.ReadSize = 200;
```

Read from the datastore twice to advance the read position to the 401st sample.

```matlab
read(DStore);
read(DStore);
```

Set ReadSize to 5 again.

```matlab
DStore.ReadSize = 5;
```

Read from the datastore.

```matlab
read(DStore)
```

ans =

5x1 timetable

<table>
<thead>
<tr>
<th>Time</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.85 sec</td>
<td>0.999</td>
</tr>
<tr>
<td>3.86 sec</td>
<td>0.99219</td>
</tr>
<tr>
<td>3.87 sec</td>
<td>0.98538</td>
</tr>
<tr>
<td>3.88 sec</td>
<td>0.97858</td>
</tr>
<tr>
<td>3.89 sec</td>
<td>0.97179</td>
</tr>
</tbody>
</table>

Extract Multiple Samples

Extract samples 1001 through 1020 (a total of 20 samples).
Reset the datastore.

```matlab
reset(DStore)
```

Advance to sample 1001.

```matlab
DStore.ReadSize = 200;
```

```matlab
for i = 1:5
    read(DStore);
end
```

Prepare to extract 20 samples from the datastore.

```matlab
DStore.ReadSize = 20;
```

Extract samples 1001 through 1020. Store the extracted data in a variable named `targetSamples`.

```matlab
targetSamples = read(DStore)
```

```
targetSamples =

    20x1 timetable

    Time       Data
    ______    ______
     9.7 sec    1.5828
     9.71 sec   1.5733
     9.72 sec   1.5664
     9.73 sec   1.5614
     9.74 sec   1.5579
     9.75 sec   1.5553
     9.76 sec   1.5703
     9.77 sec   1.582
     9.78 sec   1.5913
     9.79 sec   1.5988
     9.8 sec    1.605
     9.81 sec   1.6101
     9.82 sec   1.6145
     9.83 sec   1.6184
     9.84 sec   1.6049
     9.85 sec   1.595
```
9.86 sec  1.5877  
9.87 sec  1.5824  
9.88 sec  1.5785  
9.89 sec  1.5757  

**Find Maximum Value of Data in Datastore**

Reset the datastore.

```matlab
reset(DStore)
```

Write a `while` loop, using the `hasdata` method, to incrementally analyze the data in chunks of 200 samples.

```matlab
DStore.ReadSize = 200;
runningMax = [];
while hasdata(DStore)
    tt = read(DStore);
    rawChunk = tt.Data;
    runningMax = max([rawChunk; runningMax]);
end
```

Now, the variable `runningMax` stores the maximum value in the entire datastore.

```matlab
runningMax
```

```matlab
runningMax =
    1.6423
```

**See Also**

**Topics**

“Work with Big Data for Simulations”

**Introduced in R2017a**
**eventData**

Provide information about block method execution events

**Description**

Simulink software creates an instance of this class when a block method execution event occurs during simulation and passes it to any listeners registered for the event (see `add_exec_event_listener`). The instance specifies the type of event that occurred and the block whose method execution triggered the event. See “Access Block Data During Simulation” for more information.

**Parent**

None

**Children**

None

**Property Summary**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Type” on page 5-75</td>
<td>Type of method execution event that occurred.</td>
</tr>
<tr>
<td>“Source” on page 5-75</td>
<td>Block that triggered the event.</td>
</tr>
</tbody>
</table>
Properties

Type

Description

Type of method execution event that occurred. Possible values are:

<table>
<thead>
<tr>
<th>event</th>
<th>Occurs...</th>
</tr>
</thead>
<tbody>
<tr>
<td>'PreOutputs'</td>
<td>Before a block's Outputs method executes.</td>
</tr>
<tr>
<td>'PostOutputs'</td>
<td>After a block's Outputs method executes.</td>
</tr>
<tr>
<td>'PreUpdate'</td>
<td>Before a block's Update method executes.</td>
</tr>
<tr>
<td>'PostUpdate'</td>
<td>After a block's Update method executes.</td>
</tr>
<tr>
<td>'PreDerivatives'</td>
<td>Before a block's Derivatives method executes.</td>
</tr>
<tr>
<td>'PostDerivatives'</td>
<td>After a block's Derivatives method executes.</td>
</tr>
</tbody>
</table>

Data Type

character vector

Access

R0

Source

Description

Block that triggered the event

Data Type

Simulink.RunTimeBlock

Access

R0
LibraryBrowser.LibraryBrowser2 class

**Package:** LibraryBrowser

Simulink Library Browser

**Description**

Programmatically display, hide, size, and position the Simulink Library Browser.

**Construction**

```matlab
lb = LibraryBrowser.LibraryBrowser2
```

**Properties**

**IsOnTop — Always put library window on top**

0 (default) | 1

Always put library window on top of other Simulink Editor windows, specified as 1 for always on top.

Example: `lb.IsOnTop = 1`
# Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Meaning</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>refresh</td>
<td>Update the library browser display with changes that affect the library browser. Examples include adding a library to the library browser, removing a library from the library browser, and changes to your custom libraries, <code>slblocks</code> function, or <code>sl_customization.m</code> file. In general, refresh the library browser when you have made any changes that affect libraries on your MATLAB path that are registered in the library browser.</td>
<td><code>lb = LibraryBrowser.LibraryBrowser2; refresh(lb)</code></td>
</tr>
<tr>
<td>show</td>
<td>Display the library browser.</td>
<td><code>lb = LibraryBrowser.LibraryBrowser2; show(lb)</code></td>
</tr>
<tr>
<td>hide</td>
<td>Hide the library browser.</td>
<td><code>lb = LibraryBrowser.LibraryBrowser2; hide(lb)</code></td>
</tr>
<tr>
<td>getPosition</td>
<td>Get the position of the library browser. Returned as four integers, in pixels: upper-left x coordinate, upper-left y coordinate, width, and depth.</td>
<td><code>lb = LibraryBrowser.LibraryBrowser2; getPosition(lb)</code></td>
</tr>
<tr>
<td>ans =</td>
<td></td>
<td>50 279 600 600</td>
</tr>
<tr>
<td>setPosition</td>
<td>Set the position of the library browser. Use an array of integers, in pixels: upper-left x coordinate, upper-left y coordinate, width, and depth.</td>
<td><code>lb = LibraryBrowser.LibraryBrowser2; setPosition(lb,[70 250 500 500])</code></td>
</tr>
</tbody>
</table>
See Also

Topics
“Customize Library Browser Appearance”
“Registering Customizations”

Introduced in R2016b
allowModelReferenceDiscreteSampleTimeInheritanceImpl

Model reference sample time inheritance status for discrete sample times

Syntax

flag = allowModelReferenceDiscreteSampleTimeInheritanceImpl(obj)

Description

flag = allowModelReferenceDiscreteSampleTimeInheritanceImpl(obj) specifies whether a System object in a referenced model is allowed to inherit the sample time of the parent model. Use this method only for System objects that use discrete sample time and are intended for inclusion in Simulink via the MATLAB System block.

Run-Time Details

During model compilation, Simulink sets the referenced model sample time inheritance before the System object setupImpl method is called.

Method Authoring Tips

You must set Access = protected for this method.

Input Arguments

obj — System object
System object

System object handle used to access properties, states, and methods specific to the object. If your allowModelReferenceDiscreteSampleTimeInheritanceImpl method does not use the object, you can replace this input with ~.
Output Arguments

flag — discrete sample time inheritance flag
true | false

Flag indicating whether model reference discrete sample time inheritance is allowed for the MATLAB System block containing the System object, returned as a logical value.

The default value for this argument depends on the number of inputs to the System object. If you want to use the default value, you do not need to include this method in your System object class definition file.

<table>
<thead>
<tr>
<th>Number of System object Inputs</th>
<th>Default Value and Override Effects</th>
</tr>
</thead>
<tbody>
<tr>
<td>No inputs</td>
<td>Default: false — Model reference discrete sample time inheritance is not allowed.</td>
</tr>
<tr>
<td></td>
<td>If your System object uses discrete sample time in its algorithm, override the default by returning true from allowModelReferenceDiscreteSampleTimeInheritanceImpl.</td>
</tr>
<tr>
<td>One or more inputs</td>
<td>Default: true — If no other Simulink constraint prevents it, model reference sample time inheritance is allowed.</td>
</tr>
</tbody>
</table>

Examples

Set Sample Time Inheritance for System Object

For a System object that has one or more inputs, to disallow model reference discrete sample time inheritance for that object, set the sample time inheritance to false. Include this code in your class definition file for the object.

```matlab
methods (Access = protected)
    function flag = allowModelReferenceDiscreteSampleTimeInheritanceImpl(~)
        flag = false;
```
See Also

matlab.System

Topics

“Set Model Reference Discrete Sample Time Inheritance”
“Model Reference Basics”
“Referenced Model Sample Times”

Introduced in R2016a
getInputNamesImpl

Names of MATLAB System block input ports

Syntax

names = getInputNamesImpl(obj)

Description

names = getInputNamesImpl(obj) specifies the names of the input ports from System object, obj implemented in a MATLAB System block. The size of names matches the number of inputs returned by the getNumInputs method. If you change a property value that changes the number of inputs, the names of those inputs also change.

Class Information

This method is part of the matlab.System class.

Run-Time Details

getInputNamesImpl is called by the MATLAB System block.

Method Authoring Tips

You must set Access = protected for this method.

Input Arguments

obj — System object

System object
System object handle used to access properties, states, and methods specific to the object. If your `getInputNamesImpl` method does not use the object, you can replace this input with ~.

**Output Arguments**

`names — Names of inputs`  
empty string array (default) | string array  

Names of the inputs for the specified object, returned as a string array whose length equals the number of inputs.

**Examples**

**Specify Input and Output Port Names**

This example shows how to specify the names of ports of a MATLAB System block by adding methods to the underlying System object.

To specify port names for the MATLAB System block, you must add specific methods to the underlying System object. In the System object, add the `getInputNamesImpl` method to specify the names of two input ports as `upper` and `lower`. Additionally, add the `getOutputNamesImpl` method to specify the output port name as `Count`.

```matlab
classdef ExamplePortNames < matlab.System  
    properties(Nontunable)  
        UseOptionalInput (1,1) logical = false;  
    end  
    methods (Access = protected)  
        function inputNames = getInputNamesImpl(obj)  
            if getNumInputs(obj) > 1  
                inputNames = ["upper","lower"];  
            else  
                inputNames = "upper";  
            end  
        end  
    end  
end
```

5 Simulink Classes
function outputNames = getOutputNamesImpl(~)
    outputNames = "count";
end

function y = stepImpl(obj,upper,lower)
    if obj.UseOptionalInput == 2
        y = (upper + lower)/2;
    else
        y = upper;
    end
end

function num = getNumInputsImpl(obj)
    num = 1;
    if obj.UseOptionalInput
        num = 2;
    end
end

When you include the System object in a MATLAB System block, the ports appears as:

![Example Port Names](image)

See Also
getInputNamesImpl | getOutputNamesImpl

Topics
“Specify Input and Output Names”

Introduced in R2013b
getOutputNamesImpl

Names of MATLAB System block output ports

Syntax

names = getOutputNamesImpl(obj)

Description

names = getOutputNamesImpl(obj) specifies the names of the output ports from System object, obj implemented in a MATLAB System block. The size of names matches the number of outputs returned by the getNumOutputs method. If you change a property value that affects the number of outputs, the names of those outputs also change.

Class Information

This method is part of the matlab.System class.

Run-Time Details

getOutputNamesImpl is called by the MATLAB System block.

Method Authoring Tips

You must set Access = protected for this method.

Input Arguments

obj — System object
System object
System object handle used to access properties, states, and methods specific to the object. If your `getOutputNamesImpl` method does not use the object, you can replace this input with `~`.

**Output Arguments**

*names — Names of outputs*
empty string array (default) | string array

Names of the outputs for the specified object, returned as a string array whose length equals the number of outputs.

**Examples**

**Specify Input and Output Port Names**

This example shows how to specify the names of ports of a MATLAB System block by adding methods to the underlying System object.

To specify port names for the MATLAB System block, you must add specific methods to the underlying System object. In the System object, add the `getInputNamesImpl` method to specify the names of two input ports as `upper` and `lower`. Additionally, add the `getOutputNamesImpl` method to specify the output port name as `Count`.

classdef ExamplePortNames < matlab.System
    properties(Nontunable)
        UseOptionalInput (1,1) logical = false;
    end
    methods (Access = protected)
        function inputNames = getInputNamesImpl(obj)
            if getNumInputs(obj) > 1
                inputNames = ['upper','lower'];
            else
                inputNames = 'upper';
            end
        end
end
When you include the System object in a MATLAB System block, the ports appears as:

![Port Diagram](image)

**See Also**
getInputNamesImpl | getNumOutputsImpl

**Topics**
“Specify Input and Output Names”

**Introduced in R2013b**
getPropertyGroupsImpl

Property groups for System object display

Syntax

group = getPropertyGroupsImpl

Description

group = getPropertyGroupsImpl specifies how to display properties in the block dialog of a MATLAB System block. You can specify:

- Sections — Arrange properties into sections (panel on the MATLAB System block dialog) with `matlab.system.display.Section`.
- Section Groups — Arrange properties and sections into groups (tabs on the MATLAB System block dialog) with `matlab.system.display.SectionGroup`.
- Order of properties — Specify the order of properties within sections/groups
- Order of sections — Specify the order of sections within groups.

If you create a section, but that section is not in a section group, its properties appear above the block dialog tab panels.

Run-Time Details

In Simulink, `getPropertyGroupsImpl` modifies the MATLAB System block dialog.

In MATLAB, `getPropertyGroupsImpl` modifies how properties appear when the System object is displayed at the MATLAB command line. `getPropertyGroupsImpl` defines the order and groupings of properties. If your `getPropertyGroupsImpl` defines multiple section groups, only properties from the first section group are displayed at the command line. To display properties in other sections, a link is provided at the end of a System object property display. Group titles are also displayed at the command line.
Method Authoring Tips

- You must set `Access = protected` for this method.
- If you include a `getPropertyGroupsImpl` method but do not list all properties, the excluded properties do not appear in the dialog box.
- To omit the Main title for the first group of properties, in `matlab.system.display.SectionGroup`, set `TitleSource` to 'Auto'.
- By default if you do not customize this method in your System object, the MATLAB System block dialog displays all public properties in the order specified in the System object class file.
- To customize property names, add comments above each property. For an example, see "System Object to Block Dialog Box Default Mapping".

Output Arguments

group — Property sections and groups
array of `matlab.system.display.SectionGroup` or `matlab.system.display.Section` objects

Property groups and sections to pass to the MATLAB System block, specified as an array of `matlab.system.display.Section` and `matlab.system.display.SectionGroup` objects.

Examples

Define Property Sections

This example customizes the block dialog box for a MATLAB System block by specifying property display names and modifying the `getPropertyGroupImpl` method. This customization is demonstrated with the System object `AddPropertySections`.

Change Property Labels

To change the property label that appears on the dialog box, add comments before each property in this format `%PropertyName Block Dialog Label` with no space between the percent sign and the property name. For example, to display the `UseAlpha` property as `Use alpha`, specify:
The AddPropertySections System object included with this example relabels properties for display in the MATLAB System block dialog.

**Organize Dialog Box**

To organize the properties on the dialog box, the AddPropertySections System object class defines a `getPropertyGroupsImpl` method. Inside the `getPropertyGroupsImpl` method, this example defines two sections, each with two properties.

```matlab
classdef AddPropertySections < matlab.System
    % AddPropertySections Customized dialog with two parameter sections
    % Public, tunable properties
    properties
        %NumberOfShapes Number of shapes
        NumberOfShapes = 10
        Alpha = 0.75
    end

    % Public, nontunable properties
    properties(Nontunable)
        Coloring (1, 1) {mustBeMember(Coloring, ["red","blue","green"])} = "red"
        %UseAlpha Use alpha
        UseAlpha (1,1) logical = false
    end

    methods (Static, Access = protected)
        function groups = getPropertyGroupsImpl
            % Section for the value parameters
            valueSection = matlab.system.display.Section(...
                'Title','Shape parameters',...
                'PropertyList',{"NumberOfShapes","Coloring"});

            % Section for the threshold parameters
            shadingSection = matlab.system.display.Section(...
                'Title','Shading parameters',...
                'PropertyList',{"UseAlpha","Alpha"});
        end
    end
end
```
groups = [valueSection, shadingSection];
end
end

**Resulting Dialog Box**

```matlab
load_system('CustomSystemBlockDialog')
open_system('CustomSystemBlockDialog/MATLAB System')
```
Define Block Dialog Tabs, Sections, and Order of Properties

This example customizes the block dialog box for the `MultipleGroupsWithTabs` MATLAB System block by specifying property display names and modifying the `getPropertyGroupImpl` method.

**Change Property Label**

To change the property label that appears on the dialog box, add comments before each property in this format `%PropertyName Block Dialog Label` with no space between the comment and the property name. For example, to display the `StartValue` property as `Start Value`, specify:

```
%StartValue Start Value
StartValue = 0
```
The MultipleGroupsWithTabs System object in this example relabels each property for display in the MATLAB System block dialog.

**Organize Dialog Box**

The MutlitpleGroupsWithTabs System object class defines a `getPropertyGroupsImpl` method. Inside the `getPropertyGroupsImpl` method, this example defines two tabs (section groups) and three parameter groupings (sections).

```matlab
classdef MultipleGroupsWithTabs < matlab.System
    % MultipleGroupsWithTabs Customize block dialog with multiple tabs and parameter groups.

    % Public, tunable properties
    properties
        %StartValue Start Value
        StartValue = 0

        %EndValue End Value
        EndValue = 10

        Threshold = 1

        %BlockLimit Limit
        BlockLimit = 55
    end

    % Public Nontunable
    properties(Nontunable)
        %IC1 First initial condition
        IC1 = 0

        %IC2 Second initial condition
        IC2 = 10

        %IC3 Third initial condition
        IC3 = 100

        %UseThreshold Use threshold
        UseThreshold (1,1) logical = true
    end

    methods (Static, Access = protected)
        function groups = getPropertyGroupsImpl
            % Section to always display above any tabs.
        end
    end
```
alwaysSection = matlab.system.display.Section(...
    'Title', '', 'PropertyList', {'BlockLimit'});

% Group with no sections
initTab = matlab.system.display.SectionGroup(...
    'Title', 'Initial conditions', ...
    'PropertyList', {'IC1', 'IC2', 'IC3'});

% Section for the value parameters
valueSection = matlab.system.display.Section(...
    'Title', 'Value parameters', ...
    'PropertyList', {'StartValue', 'EndValue'});

% Section for the threshold parameters
thresholdSection = matlab.system.display.Section(...
    'Title', 'Threshold parameters', ...
    'PropertyList', {'Threshold', 'UseThreshold'});

% Group with two sections: the valueSection and thresholdSection sections
mainTab = matlab.system.display.SectionGroup(...
    'Title', 'Main', ...
    'Sections', [valueSection, thresholdSection]);

% Return an array with the group-less section, the group with % two sections, and the group with no sections.
groups = [alwaysSection, mainTab, initTab];
end
end

Resulting Dialog Box

load_system('ShowSystemBlockDialog')
open_system('ShowSystemBlockDialog/MATLAB System')
See Also
matlab.system.display.Header | matlab.system.display.Section | matlab.system.display.SectionGroup

Topics
“Add Property Groups to System Object and MATLAB System Block”
“Mapping System Object Code to MATLAB System Block Dialog Box”
Introduced in R2013b
getSimulateUsingImpl

Specify value for Simulate using parameter

Syntax

simmode = getSimulateUsingImpl

Description

simmode = getSimulateUsingImpl specifies the simulation mode of the System object implemented in a MATLAB System block. The simulation mode restricts your System object to simulation using either code generation or interpreted execution. The associated showSimulateUsingImpl method controls whether the Simulate using option is displayed on the dialog box.

Run-Time Details

getSimulateUsingImpl is called by the MATLAB System block.

Method Authoring Tips

You must set Access = protected and Static for this method.

Output Arguments

simmode — Simulation mode
"Code generation" | "Interpreted execution"

Simulation mode, returned as the string "Code generation" or "Interpreted execution". If you do not include the getSimulateUsingImpl method in your class definition file, the simulation mode is unrestricted. Depending on the value returned by the associated showSimulateUsingImpl method, the simulation mode is displayed on the dialog box as either a dropdown list or not at all.
Examples

Specify the Simulation Mode

In the class definition file of your System object, define the simulation mode to display in the MATLAB System block. To prevent Simulate using from displaying, see showSimulateUsingImpl.

```matlab
methods (Static, Access = protected)
  function simMode = getSimulateUsingImpl
    simMode = "Interpreted execution";
  end
end
```

See Also

showSimulateUsingImpl

Topics

“Control Simulation Type in MATLAB System Block”

Introduced in R2015a
getSimulinkFunctionNamesImpl

Register Simulink function names used in your System object

Syntax

names = getSimulinkFunctionNamesImpl(obj)

Description

names = getSimulinkFunctionNamesImpl(obj) specifies the Simulink function names used in the System object obj.

If you use a Simulink function in your System object, you can only call a Simulink function from the stepImpl, updateImpl, or outputImpl method.

Run-Time Details

getSimulinkFunctionNamesImpl is invoked by the MATLAB System during model compilation to obtain the list of Simulink functions being called from the System object. Simulink uses this information to resolve the function names to a Simulink function declared in a Simulink Function block. If getSimulinkFunctionNamesImpl does not return the name of a function being called in the System object methods, Simulink attempts to resolve the function to a function on the MATLAB path.

Method Authoring Tips

You must set Access = protected for this method.

Input Arguments

obj — System object
System object
System object handle used to access properties, states, and methods specific to the object. If your `getSimulinkFunctionNamesImpl` method does not use the object, you can replace this input with ~.

### Output Arguments

**names — Simulink function names**

{''} (default) | cell array of character vectors | string array

The names of the Simulink function you want to call from your System object.

### Examples

#### Syntax for Registering a Simulink Function in a System object

Register three Simulink functions that you want to call from your System object. The Simulink function names must match the name registered in the Simulink Function block.

```matlab
methods (Access = protected)
    function names = getSimulinkFunctionNamesImpl(obj)
        names = ["setPosition", "doProcessing", "timesTwo"];
    end
end
```

#### Call Simulink Functions from a MATLAB System Block

This example shows two Simulink Functions conditionally called by a MATLAB System block using the nontunable properties of the System object®.

The MATLAB System block calls one of the Simulink Functions inside two different subsystems, depending on the value of the signal coming from the Sine Wave block. If the value of the signal is less than 10, the MATLAB System block calls the `timestwo_func` Simulink Function inside the SS1 Subsystem block. If the value is larger than 10, it calls the `timesthree_func` in the SS2 Subsystem block.
Function names are defined as nontunable properties, are switched from string to functions using the `str2func` function. Then, these functions are declared as properties in the `getSimulinkFunctionNamesImpl` method.

See Also

**Classes**
matlab.System

**Blocks**
MATLAB System | Simulink Function

**Topics**
“Call Simulink Functions from MATLAB System Block”
“Simulink Functions Overview”
Introduced in R2019a
**showFiSettingsImpl**

Fixed point data type tab visibility for System objects

**Syntax**

`flag = showFiSettingsImpl`

**Description**

`flag = showFiSettingsImpl` specifies whether the Data Types tab appears on the MATLAB System block dialog box. The Data Types tab includes parameters to control processing of fixed point data the MATLAB System block. You cannot specify which parameters appear on the tab. If you implement `showFiSettingsImpl`, the simulation mode is set to code generation.

The parameters that appear on the Data Types tab, which cannot be customized, are:

- **Saturate on integer overflow** — a check box to control the action to take on integer overflow for built-in integer types. The default is that the box is checked, which indicates to saturate. This is also the default for when **Same as MATLAB** is selected as the **MATLAB System fimath** option.

- **Treat these inherited Simulink signal types as fi objects** — a pull down that indicates which inherited data types to treat as fi data types. Valid options are:
  - Fixed point (default)
  - Fixed point & integer

- **MATLAB System fimath** has two radio button options:
  - **Same as MATLAB** (default) — uses the current MATLAB fixed-point math settings.
  - **Specify Other** — enables the edit box for specifying the desired fixed-point math settings.

For information on setting fixed-point math, see **fimath**.
Run-Time Details

showFiSettingsImpl is called by the MATLAB System block. For more information, see “Simulink Engine Phases Mapped to System Object Methods”.

Method Authoring Tips

- If you do not want to display the tab, you do not need to implement this method in your class definition file.
- You must set Access = protected for this method.

Output Arguments

flag — Display Data Types tab
false (default) | true

Flag indicating whether to display the Data Types tab on the MATLAB System block mask, returned as a logical scalar value. Returning a true value displays the tab. A false value does not display the tab.

Examples

Show the Data Types Tab

Show the Data Types tab on the MATLAB System block dialog box.

methods (Static, Access = protected)
    function isVisible = showFiSettingsImpl
        isVisible = true;
    end
end

If you set the flag, isVisible, to true, the tab appears as follows when you add the object to Simulink with the MATLAB System block.
See Also

Topics
“Add Data Types Tab to MATLAB System Block”
Introduced in R2016a
showSimulateUsingImpl

Visibility of Simulate using parameter

Syntax

flag = showSimulateUsingImpl

Description

flag = showSimulateUsingImpl specifies whether Simulate using appears on the MATLAB System block dialog box.

Run-Time Details

showSimulateUsingImpl is called by the MATLAB System block.

Method Authoring Tips

You must set Access = protected and Static for this method.

Output Arguments

flag — Display parameter on block dialog box
type: logical | false

Flag indicating whether to display the Simulate using parameter and dropdown list on the MATLAB System block mask. true displays the parameter and dropdown list. false hides the parameter and dropdown list.

Examples
Hide the Simulate using Parameter

Hide the **Simulate using** parameter on the MATLAB System block dialog box.

methods (Static, Access = protected)
  function flag = showSimulateUsingImpl
    flag = false;
  end
end

If you set the flag to **true** or omit the `showSimulateUsingImpl` method, which defaults to **true**, the dialog appears as follows when you add the object to Simulink with the MATLAB System block.

If you also specify a single value for `getSimulateUsingImpl`, the dialog appears as follows when you add the object to Simulink with the MATLAB System block.
See Also
getSimulateUsingImpl

Topics
“Control Simulation Type in MATLAB System Block”

Introduced in R2015a
getGlobalNamesImpl

Global variable names for MATLAB System block

Syntax

name = getGlobalNamesImpl(obj)

Description

name = getGlobalNamesImpl(obj) specifies the names of global variables that are declared in a System object for use in a Simulink P-code file. For P-code files, in addition to declaring your global variables in stepImpl, outputImpl, or updateImpl, you must include the getGlobalNamesImpl method. You declare global variables in a cell array in the getGlobalNamesImpl method. System objects that contain these global variables are included in Simulink using a MATLAB System block. To enable a global variable in Simulink, your model also must include a Data Store Memory block with a Data Store Name that matches the global variable name.

Run-Time Details

globalNamesImpl is called by the MATLAB System block.

Method Authoring Tips

You must set Access = protected for this method.

Input Arguments

obj — System object
  System object
System object handle used to access properties, states, and methods specific to the object. If your `getGlobalNamesImpl` method does not use the object, you can replace this input with `~`.

## Output Arguments

**name — Global variable names**

*cell array of character vectors*

Cell array containing the global variable names. The elements of the cell array are character vectors.

## Examples

### Specify Global Names

Specify two global names in your class definition file.

```matlab
methods(Access = protected)
    function glnames = getGlobalNamesImpl(obj)
        glnames = {'FEE','OTHERFEE'};
    end

    function y = stepImpl(obj,u)
        global FEE
        global OTHERFEE
        y = u - FEE * obj.lastData + OTHERFEE;
        obj.lastData = u;
    end
end
```

## See Also

`outputImpl` | `stepImpl` | `updateImpl`

## Topics

“System Object Global Variables in Simulink”
Introduced in R2016b
**getHeaderImpl**

Header for System object display

**Syntax**

```matlab
header = getHeaderImpl
```

**Description**

`header = getHeaderImpl` specifies the dialog header to display on the MATLAB System block dialog box. If you do not specify the `getHeaderImpl` method, no title or text appears for the header in the block dialog box.

**Run-Time Details**

`getHeaderImpl` is called by the MATLAB System block.

**Method Authoring Tips**

You must set `Access = protected` for this method.

**Output Arguments**

- `header` — Header text
  - `matlab.system.display.Header` class

  Header text specified as a `matlab.system.display.Header` class.

**Examples**
Define Header for System Block Dialog Box

Define a header in your class definition file for the EnhancedCounter System object.

```matlab
methods (Static, Access = protected)
    function header = getHeaderImpl
        header = matlab.system.display.Header('EnhancedCounter', ...
             'Title','Enhanced Counter');
    end
end
```

See Also
gGetPropertyGroupsImpl

Topics
“Add Header to MATLAB System Block”

Introduced in R2013b
getDiscreteStateImpl

Discrete state property values

Syntax

s = getDiscreteStateImpl(obj)

Description

s = getDiscreteStateImpl(obj) returns a struct s of internal state value properties, which have the DiscreteState attribute. The field names of the struct are the object’s DiscreteState property names. To restrict or change the values returned by getDiscreteState method, you can override this getDiscreteStateImpl method.

Run-Time Details

getDiscreteStateImpl is called by the setup method. For details, see “Detailed Call Sequence” (MATLAB).

Method Authoring Tips

• You must set Access = protected for this method.
• You cannot modify any properties in this method.

Input Arguments

obj — System object
System object

System object handle used to access properties, states, and methods specific to the object. If your getDiscreteStateImpl method does not use the object, you can replace this input with ~.
Output Arguments

s — State values
struct

State values, returned as a struct.

Examples

Get Discrete State Values

Use the getDiscreteStateImpl method in your class definition file to get the discrete states of the object.

methods (Access = protected)
    function s = getDiscreteStateImpl(obj)
    end
end

See Also

setupImpl

Topics

“Define Property Attributes” (MATLAB)

Introduced in R2012b
**supportsMultipleInstanceImpl**

Support System object in Simulink For Each subsystem

**Syntax**

```matlab
flag = supportsMultipleInstanceImpl(obj)
```

**Description**

`flag = supportsMultipleInstanceImpl(obj)` specifies whether the System object can be used in a Simulink For Each subsystem via the MATLAB System block. To enable For Each support, you must include the `supportsMultipleInstanceImpl` in your class definition file and have it return `true`. Do not enable For Each support if your System object allocates exclusive resources that may conflict with other System objects, such as allocating file handles, memory by address, or hardware resources.

**Run-Time Details**

During Simulink model compilation and propagation, the MATLAB System block calls the `supportMultipleInstance` method, which then calls the `supportsMultipleInstanceImpl` method to determine For Each support.

**Method Authoring Tips**

- You must set `Access = protected` for this method.
- You cannot modify any properties in this method.

**Input Arguments**

- `obj` — System object
  System object
System object handle used to access properties, states, and methods specific to the object. If your supportsMultipleInstanceImpl method does not use the object, you can replace this input with ~.

**Output Arguments**

```plaintext
flag — Support for For Each subsystem flag
false (default) | true
```

Boolean value indicating whether the System object can be used in a For Each subsystem. The default value, if you do not include the supportMultipleInstance method, is false.

**Examples**

**Enable For-Each Support for System Object**

Specify in your class definition file that the System object can be used in a Simulink For Each subsystem.

```plaintext
methods (Access = protected)
    function flag = supportsMultipleInstanceImpl(obj)
        flag = true;
    end
end
```

**See Also**

`matlab.System`

**Topics**

“Enable For Each Subsystem Support”

**Introduced in R2014b**
processTunedPropertiesImpl

Action when tunable properties change

**Syntax**

processTunedPropertiesImpl(obj)

**Description**

processTunedPropertiesImpl(obj) specifies the algorithm to perform when one or more tunable property values change. This method is called as part of the next call to the System object after a tunable property value changes. A property is tunable only if its Nontunable attribute is false, which is the default.

**Run-Time Details**

processTunedPropertiesImpl is called when you run the System object.

In MATLAB when multiple tunable properties are changed before running the System object, processTunedPropertiesImpl is called only once for all the changes. isChangedProperty returns true for all the changed properties.

In Simulink, when a parameter is changed in a MATLAB System block dialog, the next simulation step calls processTunedPropertiesImpl before calling stepImpl. All tunable parameters are considered changed and processTunedPropertiesImpl method is called for each of them. isChangedProperty returns true for all the dialog properties.

For details, see “Detailed Call Sequence” (MATLAB).

**Method Authoring Tips**

- You must set Access = protected for this method.
- Use this method when a tunable property affects the value of a different property.
If the System object will be used in the Simulink MATLAB System block, you cannot modify any tunable properties in this method.

To check if a property has changed since `stepImpl` was last called, use `isChangedProperty` within `processTunedPropertiesImpl`. See “Specify Action When Tunable Property Changes” on page 5-122 for an example.

**Input Arguments**

`obj` — System object
System object

System object handle used to access properties, states, and methods specific to the object. If your `processTunedPropertiesImpl` method does not use the object, you can replace this input with `~`.

**Examples**

**Specify Action When Tunable Property Changes**

Use `processTunedPropertiesImpl` to recalculate the lookup table if the value of either the `NumNotes` or `MiddleC` property changes before the next call to the System object. `propChange` indicates if either property has changed.

```matlab
methods (Access = protected)
    function processTunedPropertiesImpl(obj)
        propChange = isChangedProperty(obj,'NumNotes') ||...  
        isChangedProperty(obj,'MiddleC')
        if propChange
            obj.pLookupTable = obj.MiddleC * (1+log(1:obj.NumNotes)/log(12));
        end
    end
end
```

**See Also**

`setProperties` | `validatePropertiesImpl`

**Topics**

“Process Tuned Properties” (MATLAB)
“Validate Property and Input Values” (MATLAB)
“Define Property Attributes” (MATLAB)

Introduced in R2013b
matlab.system.mixin.CustomIcon class

Package: matlab.system.mixin

Custom icon mixin class

Description

matlab.system.mixin.CustomIcon is a class that specifies the getIcon method. This method customizes the name of the icon used for the System object implemented through a MATLAB System block.

To use this method, you must subclass from this class in addition to the matlab.System base class. Type the following syntax as the first line of your class definition file, where ObjectName is the name of your object:

```
classdef ObjectName < matlab.system &...
    matlab.system.mixin.CustomIcon
```

The matlab.system.mixin.CustomIcon class is a handle class.

See Also

matlab.System | matlab.system.display.Icon

Topics

“Add Text to Block Icon”

Introduced in R2017a
**getIconImpl**

Name to display as block icon

**Syntax**

```matlab
cicon = getIconImpl(obj)
```

**Description**

`icon = getIconImpl(obj)` specifies the text or image to display on the block icon of the MATLAB System block. If you do not specify the `getIconImpl` method, the block displays the class name of the System object as the block icon. For example, if you specify `pkg.MyObject` in the MATLAB System block, the default icon is labeled `MyObject`.

**Run-Time Details**

`getIconImpl` is called by the MATLAB System block during Simulink model compilation.

**Method Authoring Tips**

You must set `Access = protected` for this method.

**Input Arguments**

- **obj — System object**
  
  System object

  System object handle used to access properties, states, and methods specific to the object. If your `getIconImpl` method does not use the object, you can replace this input with `~`. 
Output Arguments

icon — Block icon
character vector | cell array of character vectors | string | string array | matlab.system.display.Icon object

The text or image to display as the block icon. Each cell is displayed as a separate line.

Examples

Add System Block Icon Name

Specify in your class definition file the name of the block icon as 'Enhanced Counter' using two lines.

```matlab
methods (Access = protected)
    function icon = getIconImpl(~)
        icon = {'Enhanced','Counter'};
    end
end
```

Add Image to MATLAB System Block

Define an image in your class definition file.

```matlab
methods(Access = protected)
    function icon = getIconImpl(~)
        % Define icon for System block
        icon = matlab.system.display.Icon('my_icon.png');
    end
end
```

The image now appears on the System block icon.
See Also
matlab.system.display.Icon | matlab.system.mixin.CustomIcon

Topics
“Customize System Block Appearance”
“Customize MATLAB System Block Appearance”

Introduced in R2017a
matlab.system.display.Header class

Package: matlab.system.display

Header for System objects properties

Syntax

matlab.system.display.Header(N1,V1,...Nn,Vn)
matlab.system.display.Header(Obj,...)

Description

matlab.system.display.Header(N1,V1,...Nn,Vn) specifies a header for the System object, with the header properties defined in Name-Value (N,V) pairs. You use matlab.system.display.Header within the getHeaderImpl method. The available header properties are

• Title — Header title. The default value is an empty character vector.
• Text — Header description. The default value is an empty character vector.
• ShowSourceLink — Show link to source code for the object.

matlab.system.display.Header(Obj,...) creates a header for the specified System object (Obj) and sets the following property values:

• Title — Set to the Obj class name.
• Text — Set to help summary for Obj.
• ShowSourceLink — Set to true if Obj is MATLAB code. In this case, the Source Code link is displayed. If Obj is P-coded and the source code is not available, set this property to false.

You can use mfilename('class') from within this method to get the name of the System object. If you set any Name-Value pairs, those property values override the default settings.
Examples

Define System Block Header

Define a header in your class definition file.

```matlab
methods (Static, Access = protected)
    function header = getHeaderImpl
        header = matlab.system.display.Header(mfilename('class'), ...
            'Title','AlternativeTitle',...
            'Text','An alternative class description');
    end
end
```

The resulting output appears as follows. In this case, **Source code** appears because the ShowSourceLink property was set to true.

See Also

generatorImpl | `matlab.system.display.Section` | `matlab.system.display.SectionGroup`
Topics
“Classes” (MATLAB)
“Add Header to MATLAB System Block”
matlab.system.display.Section class

Package: matlab.system.display

Property group section for System objects

Syntax

matlab.system.display.Section(N1,V1,...Nn,Vn)
matlab.system.display.Section(Obj,...)

Description

matlab.system.display.Section(N1,V1,...Nn,Vn) creates a property group section for displaying System object properties, which you define using property Name-Value pairs (N,V). You use matlab.system.display.Section to define property groups using the getPropertyValueGroupsImpl method. The available Section properties are

- **Title** — Section title. The default value is an empty character vector.
- **TitleSource** — Source of section title. Valid values are 'Property' and 'Auto'. The default value is 'Property', which uses the character vector from the Title property. If the Obj name is given, the default value is Auto, which uses the Obj name.
- **Description** — Section description. The default value is an empty character vector.
- **PropertyList** — Section property list as a cell array of property names. The default value is an empty array. If the Obj name is given, the default value is all eligible display properties.

**Note** Certain properties are not eligible for display either in a dialog box or in the System object summary on the command-line. Property types that cannot be displayed are: hidden, abstract, private or protected access, discrete state, and continuous state. Dependent properties do not display in a dialog box, but do display in the command-line summary.
matlab.system.display.Section(Obj,...) creates a property group section for the specified System object (Obj) and sets the following property values:

- **TitleSource** — Set to 'Auto', which uses the Obj name.
- **PropertyList** — Set to all publicly-available properties in the Obj.

You can use `mfilename('class')` from within this method to get the name of the System object. If you set any Name-Value pairs, those property values override the default settings.

### Examples

#### Define Property Groups

Define two property groups in your class definition file by specifying their titles and property lists.

```matlab
methods (Static, Access = protected)
    function groups = getPropertyGroupsImpl
        valueGroup = matlab.system.display.Section(...
            'Title','Value parameters',...
            'PropertyList',{'StartValue','EndValue'});

        thresholdGroup = matlab.system.display.Section(...
            'Title','Threshold parameters',...
            'PropertyList',{'Threshold','UseThreshold'});
        groups = [valueGroup,thresholdGroup];
    end
end
```

When you specify the System object in the MATLAB System block, the resulting dialog box appears as follows.
See Also
getPropertyGroupsImpl | matlab.system.display.Header | matlab.system.display.SectionGroup

Topics
“Add Property Groups to System Object and MATLAB System Block”
**matlab.system.display.Action class**

**Package:** matlab.system.display

Custom button

**Syntax**

```matlab
matlab.system.display.Action(action)
matlab.system.display.Action(action,Name,Value)
```

**Description**

`matlab.system.display.Action(action)` specifies a button to display on the MATLAB System block. This button executes a function by launching a System object method or invoking any MATLAB function or code.

A typical button function launches a figure. The launched figure is decoupled from the block dialog box. Changes to the block are not synced to the displayed figure.

You define `matlab.system.display.Action` within the `getPropertyGroupsImpl` method in your class definition file. You can define multiple buttons using separate instances of `matlab.system.display.Action` in your class definition file.

`matlab.system.display.Action(action,Name,Value)` includes `Name,Value` pair arguments, which you can use to specify any properties.

**Input Arguments**

**action**

Action taken when the user presses the specified button on the MATLAB System block dialog. The action is defined as a function handle or as a MATLAB command. If you define the action as a function handle, the function definition must define two inputs. These inputs are a `matlab.system.display.ActionData` object and a System object instance, which can be used to invoke a method.
A `matlab.system.display.ActionData` object is the callback object for a display action. You use the `UserData` property of `matlab.system.display.ActionData` to store persistent data, such as a figure handle.

**Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

**Properties**

You specify these properties as part of the input using `Name,Value` pair arguments. Optionally, you can define them using `object.property` syntax.

- `ActionCalledFcn` — Action to take when the button is pressed. You cannot specify this property using a Name-Value pair argument.
- `Label` — Text to display on the button. The default value is an empty character vector.
- `Description` — Text for the button tooltip. The default value is an empty character vector.
- `Placement` — Character vector indicating where on a separate row in the property group to place the button. Valid values are `'first'`, `'last'`, or a property name. If you specify a property name, the button is placed above that property. The default value is `'last'`.
- `Alignment` — Character vector indicating how to align the button. Valid values are `'left'` and `'right'`. The default value is `'left'`.

**Examples**

**Define Button on MATLAB System Block**

Define a `Visualize` button and its associated function to open a figure that plots a ramp using the parameter values in the block dialog.

```matlab
methods(Static,Access = protected)
    function group = getPropertyGroupsImpl
```
group = matlab.system.display.Section(mfilename('class'));
group.Actions = matlab.system.display.Action(@(~,obj)...
visualize(obj),'Label','Visualize');
end
end

methods
function obj = PlotRamp(varargin)
    setProperties(obj,nargin,varargin{:});
end

function visualize(obj)
    figure;
    d = 1:obj.RampLimit;
    plot(d);
end
end

When you specify the System object in the MATLAB System block, the resulting block dialog box appears as follows.

To open the same figure, rather than multiple figures, when the button is pressed more than once, use this code instead.
methods(Static, Access = protected)
    function group = getPropertyGroupsImpl
        group = matlab.system.display.Section(mfilename('class'));
        group.Actions = matlab.system.display.Action(@(actionData, obj) ... 
            visualize(obj, actionData), 'Label', 'Visualize');
    end
end

methods
    function obj = ActionDemo(varargin)
        setProperties(obj, nargin, varargin{:});
    end

    function visualize(obj, actionData)
        f = actionData.UserData;
        if isempty(f) || ~ishandle(f)
            f = figure;
            actionData.UserData = f;
        else
            figure(f); % Make figure current
        end

        d = 1:obj.RampLimit;
        plot(d);
    end
end

See Also
getPropertyGroupsImpl | matlab.system.display.Section |
matlab.system.display.SectionGroup

Topics
"Classes" (MATLAB)
Class Attributes (MATLAB)
Property Attributes (MATLAB)
"Add Button to MATLAB System Block"
matlab.system.display.SectionGroup class

Package: matlab.system.display

Section group for System objects

Syntax

matlab.system.display.SectionGroup(N1,V1,...Nn,Vn)
matlab.system.display.SectionGroup(Obj,...)

Description

matlab.system.display.SectionGroup(N1,V1,...Nn,Vn) creates a group for displaying System object properties and display sections created with matlab.system.display.Section. You define such sections or properties using property Name-Value pairs (N,V). A section group can contain both properties and sections. You use matlab.system.display.SectionGroup to define section groups using the getPropertyGroupsImpl method. Section groups display as separate tabs in the MATLAB System block. The available Section properties are

- Title — Group title. The default value is an empty character vector.
- TitleSource — Source of group title. Valid values are 'Property' and 'Auto'. The default value is 'Property', which uses the character vector from the Title property. If the Obj name is given, the default value is Auto, which uses the Obj name. In the System object property display at the MATLAB command line, you can omit the default "Main" title for the first group of properties by setting TitleSource to 'Auto'.
- Description — Group or tab description that appears above any properties or panels. The default value is an empty character vector.
- PropertyList — Group or tab property list as a cell array of property names. The default value is an empty array. If the Obj name is given, the default value is all eligible display properties.
- Sections — Group sections as an array of section objects. If the Obj name is given, the default value is the default section for the Obj.
matlab.system.display.SectionGroup(Obj,...) creates a section group for the specified System object (Obj) and sets the following property values:

- TitleSource — Set to 'Auto'.
- Sections — Set to matlab.system.display.Section object for Obj.

You can use mfilename('class') from within this method to get the name of the System object. If you set any Name-Value pairs, those property values override the default settings.

**Examples**

**Define Block Dialog Tabs**

Define in your class definition file two tabs, each containing specific properties. For this example, you use the matlab.system.display.SectionGroup, matlab.system.display.Section, and getPropertyGroupsImpl methods.

```matlab
methods (Static, Access = protected)
    function groups = getPropertyGroupsImpl
        valueGroup = matlab.system.display.Section(...
            'Title','Value parameters',...
            'PropertyList',{StartValue','EndValue'});

        thresholdGroup = matlab.system.display.Section(...
            'Title','Threshold parameters',...
            'PropertyList',{Threshold','UseThreshold'});

        mainGroup = matlab.system.display.SectionGroup(...
            'Title','Main',...
            'Sections',[valueGroup,thresholdGroup]);

        initGroup = matlab.system.display.SectionGroup(...
            'Title','Initial conditions',...
            'PropertyList',{IC1','IC2','IC3'});

        groups = [mainGroup,initGroup];
    end
end
```

matlab.system.display.SectionGroup class

5-139
The resulting dialog appears as follows when you add the object to Simulink with the MATLAB System block.
See Also
getPropertyGroupsImpl | matlab.system.display.Header | matlab.system.display.Section

Topics
“Add Property Groups to System Object and MATLAB System Block”
matlab.system.display.Icon class

Package: matlab.system.display

Custom icon image

Syntax

icon = matlab.system.display.Icon(imageFile)

Description

icon = matlab.system.display.Icon(imageFile) sets the imageFile image as the MATLAB System block icon. To set the icon image, use the icon output argument from getIconImpl.

Input Arguments

imageFile — Image file
character array

Image file to display on the block icon, specified as a character array. If the image is not on the path, use the full path to your image file.

The image file must be in a file format supported for block masks. See “Draw Static Icon”. Example: "image.png"

Examples

Add Image to MATLAB System Block

Define an image in your class definition file.
function icon = getIconImpl(~)
    % Define icon for MATLAB System block
    icon = matlab.system.display.Icon("my_icon.png");
end
end

The image now appears on the MATLAB System block icon.

See Also
getIconImpl | matlab.system.mixin.CustomIcon

Topics
Class Attributes (MATLAB)
Property Attributes (MATLAB)
“Customize System Block Appearance”

Introduced in R2017a
matlab.system.mixin.Propagates class

Package: matlab.system.mixin

Signal characteristics propagation mixin class

Description

matlab.system.mixin.Propagates specifies the output size, data type, and complexity of a System object. Use this mixin class and its methods when you will include your System object in Simulink via the MATLAB System block. This mixin is called by the MATLAB System block during Simulink model compilation.

Note If your System object has exactly one input and one output and no discrete property states, or if you do not need bus support, you do not have to implement any of these methods. The matlab.system.mixin.Propagates provides default values in these cases.

Implement the methods of this class when Simulink cannot infer the output specifications directly from the inputs or when you want bus support. If you do not include this mixin, Simulink cannot propagate the output or bus data type, an error occurs.

To use this mixin, subclass from this matlab.system.mixin.Propagates in addition to subclassing from the matlab.System base class. Type the following syntax as the first line of your class definition file. ObjectName is the name of your System object.

```matlab
classdef ObjectName < matlab.System &...
    matlab.system.mixin.Propagates
```

The matlab.system.mixin.Propagates class is a handle class.

See Also

matlab.System
Topics
“Set Output Data Type”
“Set Output Size”
“Set Output Complexity”
“Set Fixed- or Variable-Size Output”
“Set Discrete State Output Specification”

Introduced in R2014a
getDiscreteStateSpecificationImpl

Discrete state size, data type, and complexity

Syntax

[size, dataType, complexity] = getDiscreteStateSpecificationImpl(obj, propertyName)

Description

[size, dataType, complexity] = getDiscreteStateSpecificationImpl(obj, propertyName) returns the size, data type, and complexity of the discrete state property. This property must be a discrete state property. You must define this method if your System object has discrete state properties and is used in the MATLAB System block.

Run-Time Details

getDiscreteStateSpecificationImpl is called by the MATLAB System block during Simulink model compilation.

Method Authoring Tips

• You must set Access = protected for this method.
• You cannot modify any properties in this method.

Input Arguments

obj — System object
    System object
System object handle used to access properties, states, and methods specific to the object. If your `getDiscreteStateSpecificationImpl` method does not use the object, you can replace this input with ~.

`propertyName` — Discrete state property name

Name of discrete state property of the System object

**Output Arguments**

`size` — Property size

[1 1] (default) | numeric vector

Vector containing the length of each dimension of the property.

`dataType` — Property data type

`double` (default) | character vector

Data type of the property. For built-in data types, `dataType` is a character vector. For fixed-point data types, `dataType` is a numeric type object.

`complexity` — Complexity of the property

false (default) | true

Complexity of the property as a scalar, logical value:

- true — complex
- false — real

**Examples**

**Specify Discrete State Property Size, Data Type, and Complexity**

Specify in your class definition file the size, data type, and complexity of a discrete state property.

```
methods (Access = protected)
    function [sz,dt,cp] = getDiscreteStateSpecificationImpl(~,name)
```

sz = [1 1];
dt = 'double';
cp = false;
end
end

See Also

Topics
“Set Discrete State Output Specification”

Introduced in R2013b
getOutputDataTypeImpl

Data types of output ports

Syntax

\[ [dt_1, dt_2, \ldots, dt_n] = \text{getOutputDataTypeImpl}(\text{obj}) \]

Description

\[ [dt_1, dt_2, \ldots, dt_n] = \text{getOutputDataTypeImpl}(\text{obj}) \]
returns the data type of each output port as a character vector for built-in data types or as a numeric object for fixed-point data types. The number of outputs must match the value returned from the \text{getNumOutputsImpl} method or the number of output arguments listed in the \text{stepImpl} method.

For System objects with one input and one output and where you want the input and output data types to be the same, you do not need to implement this method. In this case, \text{getOutputDataTypeImpl} assumes the input and output data types are the same and returns the data type of the input.

If your System object has more than one input or output, and you use propagation, you must set the output data types in the \text{getOutputDataTypeImpl} method. For Simulink, if the input and output data types are different, you might have to cast the output value to the data type of the appropriate \text{dt}_n output argument. You specify this casting in the \text{stepImpl} method. For bus output, you must specify the name of the output bus in \text{getOutputDataTypeImpl}.

If needed to determine the output data type, you can use \text{propagatedInputDataType} within the \text{getOutputDataTypeImpl} method to obtain the input type.

Run-Time Details

getOutputDataTypeImpl is called by the MATLAB System block.
Method Authoring Tips

• You must set Access = protected for this method.
• You cannot modify any properties in this method.
• If you are debugging your code and examine the data types before Simulink completes propagation, you might see outputs with empty, [ ], data types. This occurs because Simulink has not completed setting the output data types.

Input Arguments

obj — System object
System object

System object handle used to access properties, states, and methods specific to the object. If your getOutputDataTypeImpl method does not use the object, you can replace this input with ~.

Output Arguments

dt_1,dt_2,... — Property data type
character vector | numeric type object

Data type of the property. For built-in data types, dt is a character vector. For fixed-point data types, dt is a numeric type object.

Examples

Specify Output Data Type

Specify, in your class definition file how to control the output data type from a MATLAB System block. This example shows how to use the getOutputDataTypeImpl method to change the output data type from single to double, or propagate the input as a double. It also shows how to cast the data type to change the output data type in the stepImpl method.
```matlab
classdef DataTypeChange < matlab.System & ...
    matlab.system.mixin.Propagates

    properties (Nontunable)
        Quantize = false
    end

    methods (Access = protected)
        function y = stepImpl(obj,u)
            if obj.Quantize == true
                % Cast for output data type to differ from input.
                y = single(u);
            else
                % Propagate output data type.
                y = u;
            end
        end

        function out = getOutputDataTypeImpl(obj)
            if obj.Quantize == true
                out = "single";
            else
                out = propagatedInputDataType(obj,1);
            end
        end
    end
end
end

classdef DataTypeChange < matlab.System & ...
    matlab.system.mixin.Propagates

    properties (Nontunable)
        Quantize = false
    end

    methods (Access = protected)
        function y = stepImpl(obj,u)
            if obj.Quantize == true
                % Cast for output data type to differ from input.
                y = single(u);
            else
                % Propagate output data type.
                y = u;
            end
        end

        function out = getOutputDataTypeImpl(obj)
            if obj.Quantize == true
                out = "single";
            else
                out = propagatedInputDataType(obj,1);
            end
        end
    end
end

Specify Bus Output

Specify, in your class definition file, that the System object data type is a bus. You must also include a property to specify the bus name.

properties (Nontunable)
    OutputBusName = "myBus";
end

methods (Access = protected)
    function out = getOutputDataTypeImpl(obj)
        out = obj.OutputBusName;
    end
end
end
```

Specify Bus Output

Specify, in your class definition file, that the System object data type is a bus. You must also include a property to specify the bus name.

properties (Nontunable)
    OutputBusName = "myBus";
end

methods (Access = protected)
    function out = getOutputDataTypeImpl(obj)
        out = obj.OutputBusName;
    end
end
```
See Also
propagatedInputDataType

Topics
“Set Output Data Type”

Introduced in R2013b
**getOutputSizeImpl**

Sizes of output ports

**Syntax**

\[sz_1,sz_2,...,sz_n\] = getOutputSizeImpl(obj)

**Description**

\[sz_1,sz_2,...,sz_n\] = getOutputSizeImpl(obj) returns the size of each output port. The number of outputs must match the value returned from the `getNumOutputs` method or the number of output arguments listed in the `stepImpl` method.

If your System object has only one input and one output and you want the input and output sizes to be the same, you do not need to implement this method. In this case `getOutputSizeImpl` assumes that the input and output sizes are the same and returns the size of the input. For variable-size inputs in MATLAB, the size varies each time you run your object. For variable-size inputs in Simulink, the output size is the maximum input size.

You must implement the `getOutputSizeImpl` method to define the output size, if:

- Your System object has more than one input or output
- You need the output and input sizes to be different.

  If the output size differs from the input size, you must also use the `propagatedInputSize` method

**Run-Time Details**

During Simulink model compilation and propagation, the MATLAB System block calls the `getOutputSizeImpl` method to determine the output size.

All inputs default to variable-size inputs. For these inputs, the output size is the maximum input size.
Method Authoring Tips

• You must set Access = protected for this method.
• In this method, you cannot modify any properties.

Input Arguments

obj — System object
System object

System object handle used to access properties, states, and methods specific to the object. If your getOutputSizeImpl method does not use the object, you can replace this input with ~.

Output Arguments

sz_1,sz_2,... — Output size
numeric vector

Vector containing the size of each output port.

Examples

Specify Output Size

Specify in your class definition file the size of a System object output.

methods (Access = protected)
    function sz_1 = getOutputSizeImpl(obj)
        sz_1 = [1 1];
    end
end
Specify Multiple Output Ports

Specify in your class definition file the sizes of multiple System object outputs.

methods (Access = protected)
    function [sz_1,sz_2] = getOutputSizeImpl(obj)
        sz_1 = propagatedInputSize(obj,1);
        sz_2 = [1 1];
    end
end

Specify Output When Using Propagated Input Size

Specify in your class definition file the size of System object output when it depends on the propagated input size.

methods (Access = protected)
    function varargout = getOutputSizeImpl(obj)
        varargout{1} = propagatedInputSize(obj,1);
        if obj.HasSecondOutput
            varargout{2} = [1 1];
        end
    end
end

See Also

propagatedInputSize

Topics

“Set Output Size”

Introduced in R2013b
isOutputComplexImpl

Complexity of output ports

Syntax

\[ [cp_1, cp_2, \ldots, cp_n] = \text{isOutputComplexImpl}(\text{obj}) \]

Description

\[ [cp_1, cp_2, \ldots, cp_n] = \text{isOutputComplexImpl}(\text{obj}) \] returns whether each output port has complex data. The number of outputs must match the value returned from the `getNumOutputs` method or the number of output arguments listed in the `stepImpl` method.

For System objects with one input and one output and where you want the input and output complexities to be the same, you do not need to implement this method. In this case `isOutputComplexImpl` assumes the input and output complexities are the same and returns the complexity of the input.

If your System object has more than one input or output or you need the output and input complexities to be different, you must implement the `isOutputComplexImpl` method to define the output complexity. You also must use the `propagatedInputComplexity` method if the output complexity differs from the input complexity.

Run-Time Details

During Simulink model compilation and propagation, the MATLAB System block calls the `isOutputComplex` method, which then calls the `isOutputComplexImpl` method to determine the output complexity.

Method Authoring Tips

- You must set `Access = protected` for this method.
- You cannot modify any properties in this method.
Input Arguments

\texttt{obj} — System object

System object handle used to access properties, states, and methods specific to the object. If your \texttt{isOutputComplexImpl} method does not use the object, you can replace this input with \texttt{~}.

Output Arguments

\texttt{cp_1, cp_2, \ldots} — Complexity flag

\texttt{true|false}

Logical, scalar value indicating whether the specific output port is complex (\texttt{true}) or real (\texttt{false}).

Examples

Specify Output as Real-Valued

Specify in your class definition file that the output from a System object is a real value.

\begin{verbatim}
methods (Access = protected)
    function c1 = isOutputComplexImpl(obj)
        c1 = false;
    end
end
\end{verbatim}

See Also

\texttt{propagatedInputComplexity}

Topics

“Set Output Complexity”
Introduced in R2013b
isOutputFixedSizeImpl

Fixed- or variable-size output ports

Syntax

[flag_1,flag_2,...flag_n] = isOutputFixedSizeImpl(obj)

Description

[flag_1,flag_2,...flag_n] = isOutputFixedSizeImpl(obj) returns whether each output port is fixed size. The number of outputs must match the value returned from the getNumOutputs method, which is the number of output arguments listed in the stepImpl method.

For System objects with one input and one output and where you want the input and output fixed sizes to be the same, you do not need to implement this method. In this case isOutputFixedSizeImpl assumes the input and output fixed sizes are the same and returns the fixed size of the input.

If your System object has more than one input or output or you need the output and input fixed sizes to be different, you must implement the isOutputFixedSizeImpl method to define the output fixed size. You also must use the propagatedInputFixedSize method if the output fixed size status differs from the input fixed size status.

Run-Time Details

During Simulink model compilation and propagation, the MATLAB System block calls the isOutputFixedSize method, which then calls the isOutputFixedSizeImpl method to determine the output fixed size.

All inputs default to variable-size inputs For these inputs, the output size is the maximum input size.
**Method Authoring Tips**

- You must set `Access = protected` for this method.
- You cannot modify any properties in this method.

**Input Arguments**

`obj` — System object  
System object  

System object handle used to access properties, states, and methods specific to the object. If your `isOutputFixedSizeImpl` method does not use the object, you can replace this input with ~.

**Output Arguments**

`flag_1, flag2, ...` — Fixed size flag  
`true` | `false`  

Logical, scalar value indicating whether the specific output port is fixed size (`true`) or variable size (`false`).

**Examples**

**Specify Output as Fixed Size**

Specify in your class definition file that the output from a System object is of fixed size.

```matlab  
methods (Access = protected)  
    function c1 = isOutputFixedSizeImpl(obj)  
    c1 = true;  
```
See Also
propagatedInputFixedSize

Topics
“Set Fixed- or Variable-Size Output”

Introduced in R2013b
propagatedInputComplexity

Complexity of input during Simulink propagation

Syntax

flag = propagatedInputComplexity(obj,index)

Description

flag = propagatedInputComplexity(obj,index) returns true or false to indicate whether the input argument for the indicated System object is complex. index specifies the input for which to return the complexity flag.

You can use propagatedInputComplexity only from within the isOutputComplexImpl method in your class definition file. Use isOutputComplexImpl when:

- Your System object has more than one input or output.
- The input complexity determines the output complexity.
- The output complexity must differ from the input complexity.

Input Arguments

obj — System object
System object

System object handle used to access properties, states, and methods specific to the object. If your propagatedInputComplexity method does not use the object, you can replace this input with ~.

index — Input index
positive integer
Index of the specified input. Do not count the obj in the index. The first input is always obj.

**Output Arguments**

flag — Complexity flag
```
true | false
```
Complexity of the specified input, returned as true or false

**Examples**

**Match Input and Output Complexity**

Get the complexity of the second input when you run the object and set the output to match it. Assume that the first input has no impact on the output complexity.

```matlab
methods (Access = protected)
    function outcomplx = isOutputComplexImpl(obj)
        outcomplx = propagatedInputComplexity(obj,2);
    end
end
```

**See Also**

isOutputComplexImpl

**Topics**

“Set Output Complexity”

**Introduced in R2014a**
propagatedInputDataType

Data type of input during Simulink propagation

Syntax

\[ dt = \text{propagatedInputDataType}(obj,\text{index}) \]

Description

\[ dt = \text{propagatedInputDataType}(obj,\text{index}) \]

returns the data type of an input argument for a System object. \text{index} specifies the input for which to return the data type.

You can use \text{propagatedInputDataType} only from within \text{getOutputDataTypeImpl}. Use \text{getOutputDataTypeImpl} when:

• Your System object has more than one input or output.
• The input data type status determines the output data type.
• The output data type must differ from the input data type.

Input Arguments

\textbf{obj — System object}

System object

System object handle used to access properties, states, and methods specific to the object. If your \text{propagatedInputDataType} method does not use the object, you can replace this input with \text{~}.

\textbf{index — Input index}

positive integer

Index of the specified input. Do not count the \text{obj} in the \text{index}. The first input is always \text{obj}. 

5-164
Output Arguments

**dt — Input data type**
character vector | numerictype object

Data type of the specified input, returned as a character vector for floating-point input or as a numerictype for fixed-point input.

Examples

**Match Input and Output Data Type**

Get the data type of the second input. If the second input data type is `double`, then the output data type is `int32`. For all other cases, the output data type matches the second input data type. Assume that the first input has no impact on the output.

```matlab
methods (Access = protected)
    function dt = getOutputDataTypeImpl(obj)
        if strcmpi(propagatedInputDataType(obj,2),'double')
            dt = 'int32';
        else
            dt = propagatedInputDataType(obj,2);
        end
    end
end
```

See Also

getOutputDataTypeImpl

Topics

“Set Output Data Type”
“Data Type Propagation”

Introduced in R2014a
propagatedInputFixedSize

Fixed-size status of input during Simulink propagation

Syntax

flag = propagatedInputFixedSize(obj,index)

Description

flag = propagatedInputFixedSize(obj,index) returns true or false to indicate whether an input argument of a System object is fixed size. index specifies the input for which to return the fixed-size flag.

You can use propagatedInputFixedSize only from within isOutputFixedSizeImpl. Use isOutputFixedSizeImpl when:

- Your System object has more than one input or output.
- The input fixed-size status determines the output fixed-size status.
- The output fixed-size status must differ from the input fixed-size status.

Input Arguments

obj — System object

System object

System object handle used to access properties, states, and methods specific to the object. If your propagatedInputFixedSize method does not use the object, you can replace this input with ~.

index — Input index

positive integer

Index of the specified input. Do not count the obj in the index. The first input is always obj.
Output Arguments

flag — Fixed size flag
    true | false

Fixed-size status of the specified input, returned as true or false.

Examples

Match Fixed-Size Status of Input and Output

Get the fixed-size status of the third input and set the output to match it. Assume that the first and second inputs have no impact on the output.

```matlab
methods (Access = protected)
    function outtype = isOutputFixedSizeImpl(obj)
        outtype = propagatedInputFixedSize(obj,3)
    end
end
```

See Also

isOutputFixedSizeImpl

Topics

“Set Fixed- or Variable-Size Output”

Introduced in R2014a
propagatedInputSize

Size of input during Simulink propagation

Syntax

size = propagatedInputSize(obj,index)

Description

size = propagatedInputSize(obj,index) returns, as a vector, the input size of the specified System object. The index specifies the input for which to return the size information. (Do not count the obj in the index. The first input is always obj.)

You can use propagatedInputSize only from within the getOutputSizeImpl method in your class definition file. Use getOutputSizeImpl when:

• Your System object has more than one input or output.
• The input size determines the output size.
• The output size must differ from the input size.

Note For variable-size inputs, the propagated input size from propagatedInputSize differs depending on the environment.

• MATLAB — propagatedInputSize returns the size of the inputs used when you run the object for the first time.
• Simulink — propagatedInputSize returns the upper bound of the input sizes.

Input Arguments

obj — System object
System object
System object handle used to access properties, states, and methods specific to the object. If your `propagatedInputSize` method does not use the object, you can replace this input with ~.

**index — Input index**

integer

Index of the specified input

**Output Arguments**

**size — Input size**

numeric vector

Size of the specified input, returned as a vector

**Examples**

**Match Size of Input and Output**

Get the size of the second input. If the first dimension of the second input has a size greater than 1, then set the output size to a 1 x 2 vector. For all other cases, the output is a 2 x 1 matrix. Assume that the first input has no impact on the output size.

```matlab
methods (Access = protected)
    function outsize = getOutputSizeImpl(obj)
        size = propagatedInputSize(obj,2);
        if size(1) == 1
            outsize = [1,2];
        else
            outsize = [2,1];
        end
    end
end
```

**See Also**

`getOutputSizeImpl`

5-169
Topics
“Set Output Size”

Introduced in R2014a
matlab.system.mixin.Nondirect class

Package: matlab.system.mixin

Nondirect feedthrough mixin class

Description

matlab.system.mixin.Nondirect is a class that uses the output and update methods to process nondirect feedthrough data through a System object.

For System objects that use direct feedthrough, the object’s input is needed to generate the output at that time. For these direct feedthrough objects, running the System object calculates the output and updates the state values. For nondirect feedthrough, however, the object’s output depends only on the internal states at that time. The inputs are used to update the object states. For these objects, calculating the output with outputImpl is separated from updating the state values with updateImpl. If you use the matlab.system.mixin.Nondirect mixin and include the stepImpl method in your class definition file, an error occurs. In this case, you must include the updateImpl and outputImpl methods instead.

The following cases describe when System objects in Simulink use direct or nondirect feedthrough.

- System object supports code generation and does not inherit from the Propagates mixin — Simulink automatically infers the direct feedthrough settings from the System object code.
- System object supports code generation and inherits from the Propagates mixin — Simulink does not automatically infer the direct feedthrough settings. Instead, it uses the value returned by the isInputDirectFeedthroughImpl method.
- System object does not support code generation — Default isInputDirectFeedthroughImpl method returns false, indicating that direct feedthrough is not enabled. To override the default behavior, implement the isInputDirectFeedthroughImpl method in your class definition file.

Use the Nondirect mixin to allow a System object to be used in a Simulink feedback loop. A delay object is an example of a nondirect feedthrough object.
To use this mixin, you must subclass from this class in addition to subclassing from the `matlab.System` base class. Type the following syntax as the first line of your class definition file, where `ObjectName` is the name of your object:

```matlab
classdef ObjectName < matlab.system & matlab.system.mixin.Nondirect
```

**See Also**

`matlab.System`

**Topics**

“Use Update and Output for Nondirect Feedthrough”

**Introduced in R2013b**
isInputDirectFeedthroughImpl

Direct feedthrough status of input

Syntax

[flag1,...,flagN] = isInputDirectFeedthroughImpl(obj)
[flag1,...,flagN] = isInputDirectFeedthroughImpl(obj,
input,input2,...)

Description

[flag1,...,flagN] = isInputDirectFeedthroughImpl(obj) specifies whether
each input is a direct feedthrough input. If direct feedthrough is true, the output
deeps on the input at each time instant.

[flag1,...,flagN] = isInputDirectFeedthroughImpl(obj,
input,input2,...) uses one or more of the System object input specifications to
determine whether inputs have direct feedthrough.

If you do not include the isInputDirectFeedthroughImpl method in your System
object class definition file, all inputs are assumed to be direct feedthrough.

Code Generation

The following cases describe when System objects in Simulink code generation use direct
or nondirect feedthrough.

<table>
<thead>
<tr>
<th>System object's code generation support</th>
<th>Uses a propagation Impl method</th>
<th>SimulinkCode Generation Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y</td>
<td>N</td>
<td>Simulink automatically infers the direct feedthrough settings from the System object code.</td>
</tr>
</tbody>
</table>
~ System object's code generation support | Uses a propagation Impl method | SimulinkCode Generation Result |
--- | --- | --- |
Y | Y | Simulink does not automatically infer the direct feedthrough settings. Instead, it uses the value returned by the `isInputDirectFeedthroughImpl` method. |
N | — | Default `isInputDirectFeedthroughImpl` method returns false, indicating that direct feedthrough is not enabled. To override the default behavior, implement the `isInputDirectFeedthroughImpl` method in your class definition file. |

### Run-Time Details

`isInputDirectFeedthroughImpl` is called by the MATLAB System block.

### Method Authoring Tips

- You must set `Access = protected` for this method.
- You cannot modify, implement, or access tunable properties in this method.

### Input Arguments

**obj — System object**

System object

System object handle used to access properties, states, and methods specific to the object. If your `isInputDirectFeedthroughImpl` method does not use the object, you can replace this input with `~`.

**input1,input2,... — Inputs to the System object**

Inputs to the System object algorithm

Inputs to the algorithm (`stepImpl`) of the System object. The inputs list must match the order of inputs in the `stepImpl` signature.
Output Arguments

flag1,...,flagN — Output flag for each input to the System object
logical

Logical value, either true or false indicating whether the input is direct feedthrough. The number of output flags must match the number of inputs to the System object (inputs to stepImpl, outputImpl, or updateImpl).

Examples

Specify Input as Nondirect Feedthrough

Use isInputDirectFeedthroughImpl in your class definition file for marking all inputs as nondirect feedthrough.

```matlab
methods (Access = protected)
    function flag = isInputDirectFeedthroughImpl(~)
        flag = false;
    end
end
```

Complete Class Definition

```matlab
classdef intDelaySysObj < matlab.System
    % intDelaySysObj Delay input by specified number of samples.

    properties
        InitialOutput = 0;
    end
    properties (Nontunable)
        NumDelays = 1;
    end
    properties (DiscreteState)
        PreviousInput;
    end

    methods (Access = protected)
        function validatePropertiesImpl(obj)
            if ((numel(obj.NumDelays)>1) || (obj.NumDelays <= 0))
                error('Number of delays must be > 0 scalar value.');
            end
        end
    end
end
```
if (numel(obj.InitialOutput) > 1)
    error('Initial Output must be scalar value.');
end

function setupImpl(obj)
    obj.PreviousInput = ones(1, obj.NumDelays) * obj.InitialOutput;
end

function resetImpl(obj)
    obj.PreviousInput = ones(1, obj.NumDelays) * obj.InitialOutput;
end

function [y] = outputImpl(obj, ~)
    y = obj.PreviousInput(end);
end

function updateImpl(obj, u)
    obj.PreviousInput = [u obj.PreviousInput(1:end-1)];
end

function flag = isInputDirectFeedthroughImpl(~)
    flag = false;
end

See Also
- outputImpl
- updateImpl

Topics
- “Use Update and Output for Nondirect Feedthrough”

Introduced in R2013b
outputImpl

Output calculation from input or internal state of System object

Syntax

\[ [y_1, y_2, \ldots, y_N] = \text{outputImpl}(obj, u_1, u_2, \ldots, u_N) \]

Description

\[ [y_1, y_2, \ldots, y_N] = \text{outputImpl}(obj, u_1, u_2, \ldots, u_N) \]

specifies the algorithm to output the System object states. The output values are calculated from the states and property values. Any inputs that you set to nondirect feedthrough are ignored during output calculation.

Run-Time Details

\text{outputImpl} is called by the internal \text{output} method. It is also called before the \text{updateImpl} method. For sink objects, calling \text{updateImpl} before \text{outputImpl} locks the object. For all other types of objects, calling \text{updateImpl} before \text{outputImpl} causes an error.

Method Authoring Tips

- You must set Access = protected for this method.
- If the System object will be used in the Simulink MATLAB System block, you cannot modify any tunable properties in this method.

Input Arguments

\textit{obj} — System object

System object
System object handle used to access properties, states, and methods specific to the object. If your outputImpl method does not use the object, you can replace this input with ~.

\[u_1, u_2, \ldots u_N\] — Inputs to the algorithm

Inputs from the algorithm. The number of inputs must match the number of inputs returned by the getNumInputs method. Nondirect feedthrough inputs are ignored during normal execution of the System object. However, for code generation, you must provide these inputs even if they are empty.

**Output Arguments**

\[y_1, y_2, \ldots y_N\] — Outputs of the algorithm

Outputs calculated from the specified algorithm. The number of outputs must match the number of outputs returned by the getNumOutputs method.

**Examples**

**Set Up Output that Does Not Depend on Input**

Specify in your class definition file that the output does not directly depend on the current input with the outputImpl method. PreviousInput is a property of the obj.

```matlab
methods (Access = protected)
    function [y] = outputImpl(obj, ~)
        y = obj.PreviousInput(end);
    end
end
```

**See Also**

isNewDirectFeedthroughImpl | updateImpl
**Topics**

“Use Update and Output for Nondirect Feedthrough”

**Introduced in R2013b**
updateImpl

Update object states based on inputs

Syntax

updateImpl(obj,input1,input2,...)

Description

updateImpl(obj,input1,input2,...) specifies the algorithm to update the System object states. You implement this method when your algorithm outputs depend only on the object’s internal state and internal properties.

Run-Time Details

updateImpl is called by the update method and after the outputImpl method.

For sink objects, calling updateImpl before outputImpl locks the object. For all other types of objects, calling updateImpl before outputImpl causes an error.

Method Authoring Tips

• Do not use this method to update the outputs from the inputs.
• You must set Access = protected for this method.
• If the System object will be used in the Simulink MATLAB System block, you cannot modify any tunable properties in this method.

Input Arguments

obj — System object
    System object
System object handle used to access properties, states, and methods specific to the object. If your updateImpl method does not use the object, you can replace this input with `~`.

`input1,input2,...` — Inputs to the System object

inputs to the System object

List the inputs to the System object. The order of inputs must match the order of inputs defined in the stepImpl method.

**Examples**

**Set Up Output that Does Not Depend on Current Input**

Update the object with previous inputs. Use `updateImpl` in your class definition file. This example saves the `u` input and shifts the previous inputs.

```matlab
methods (Access = protected)
    function updateImpl(obj,u)
        obj.PreviousInput = [u obj.PreviousInput(1:end-1)];
    end
end
```

**See Also**

`isInputDirectFeedthroughImpl` | `outputImpl`

**Topics**

“Use Update and Output for Nondirect Feedthrough”

**Introduced in R2013b**
**matlab.system.mixin.SampleTime class**

Control sample time for System objects in Simulink

**Description**

`matlab.system.mixin.SampleTime` specifies the sample time specifications for a System object when it is included in a MATLAB System block. Inherit from this mixin class and use its methods to control the sample time of your System object in Simulink, via the MATLAB System block.

With this mixin, you can:

- Specify the sample time type
- Specify the sample time
- Customize the sample time with offsets and tick times
- Get the current simulation time

System objects that inherit from this mixin class must also inherit from `matlab.System`. For example:

```matlab
classdef MySystemObject < matlab.System & matlab.system.mixin.SampleTime
```

The `matlab.system.mixin.SampleTime` class is a handle class.

**Methods**

**See Also**

**Classes**
- `matlab.System`

**Blocks**
- MATLAB System
Topics
“Specify Sample Time for MATLAB System Block System Objects”

Introduced in R2017b
getSampleTimeImpl

Specify sample time type, offset time, and sample time

Syntax

sts = getSampleTimeImpl(obj)

Description

sts = getSampleTimeImpl(obj) returns the sample time specification created within the method body, sts, for the System object obj. Specify the sample time specification within the body of getSampleTimeImpl by calling createSampleTime. The sample time specification affects the simulation time when the System object is included in a MATLAB System block.

Run-Time Details

getSampleTimeImpl is called during setup by setupImpl.

Method Authoring Tips

You must set Access = protected for this method.

Default Behavior

If you do not include this method in your System object definition, the sample time is inherited.

Input Arguments

obj — System object
System object
System object handle used to access properties, states, and methods specific to the object. If your `getSampleTimeImpl` method does not use the object, you can replace this input with ~.

### Output Arguments

**sts — Sample time specification object**

Sample time specification object

An object defining the sample time specification values. You create this object with the `createSampleTime` function.

### Examples

#### Specify Inherited Sample Time

Specify that the MATLAB System block should inherit the sample from upstream blocks, except if the sample time type is controllable.

```matlab
function sts = getSampleTimeImpl(obj)
    sts = createSampleTime(obj,'ErrorOnPropagation','Controllable');
end
```

#### Specify Discrete Sample Time

Specify a discrete sample time for the MATLAB System block.

```matlab
function sts = getSampleTimeImpl(obj)
    sts = createSampleTime(obj,'Type','Discrete',
                             'SampleTime',10.2,'OffsetTime',0.5);
end
```

### See Also

`createSampleTime` | `getCurrentTime` | `getSampleTime`

### Topics

“Specify Sample Time for MATLAB System Block System Objects”
Introduced in R2017b
getSampleTime

Query sample time

Syntax

sts = getSampleTime(obj)

Description

sts = getSampleTime(obj) returns the sample time specification for the System object obj when the System object is included in a MATLAB System block. You can call getSampleTime in the stepImpl method to change the algorithm based on the sample time.

Before sample time has propagated throughout the MATLAB System block model, getSampleTime returns the getSampleTimeImpl sample time specification. If your system object does not override getSampleTimeImpl, the default Inherited sample time specification is returned.

After sample time has propagated, getSampleTime returns the sample time specification populated with the actual MATLAB System block sample time type, sample time, and offset time.

Input Arguments

obj — System object
System object

System object included in a MATLAB System block that you want to query.
Output Arguments

sts — Sample time specification object
sample time specification object

The sample time specification for the System object. For more details about sample time specification objects, see `createSampleTime`.

Examples

Return Sample Time

This example of `stepImpl` returns a count value `y`, the current simulation time `ct`, and the sample time `st`. The sample time is obtained by calling `getSampleTime`.

```matlab
function [y,ct,st] = stepImpl(obj,u)
    y = obj.Count + u;
    obj.Count = y;
    ct = getCurrentTime(obj);
    sts = getSampleTime(obj);
    st = sts.SampleTime;
end
```

For a complete class definition, see “Specify Sample Time for MATLAB System Block System Objects”.

See Also

`createSampleTime` | `getCurrentTime` | `getSampleTimeImpl`

Topics

“Specify Sample Time for MATLAB System Block System Objects”

Introduced in R2017b
getCurrentTime

Current simulation time in MATLAB System block

Syntax

\[
time = \text{getCurrentTime}(obj)
\]

Description

\[
time = \text{getCurrentTime}(obj)
\]
returns the current simulation time in the MATLAB System block. Call this method in the `stepImpl` method of your System object.

Dependencies

If the MATLAB System block is operating in continuous sample time, `getCurrentTime` may return non-monotonic times due to solver operation.

Input Arguments

\[
obj \quad \text{— System object}
\]

system object

System object included in a MATLAB System block that you want to query.

Output Arguments

\[
time \quad \text{— Current simulation time}
\]

double

The current simulation time of the MATLAB System block that contains the System object.
Examples

Return Current Simulation Time

This example of function `stepImpl` returns a count value `y` and the current simulation time `ct`. The simulation time is obtained by calling `getCurrentTime`.

```matlab
function [y,ct] = stepImpl(obj,u)
    y = obj.Count + u;
    obj.Count = y;
    ct = getCurrentTime(obj);
end
```

For a complete class definition, see “Specify Sample Time for MATLAB System Block System Objects”.

See Also

`createSampleTime` | `getSampleTime` | `getSampleTimeImpl`

Topics

“Specify Sample Time for MATLAB System Block System Objects”

Introduced in R2017b
createSampleTime

Create sample time specification object

Syntax

sts = createSampleTime(obj)
sts = createSampleTime(obj,'Type',Type)
sts = createSampleTime(obj,'Type',Type,Name,Value)

Description

sts = createSampleTime(obj) creates a sample time specification object for inherited sample time for the System object obj. Use this sample time specification object in the getSampleTimeImpl method of your System object. The sample time specification affects the simulation time when the System object is included in a MATLAB System block. If you do not include this method in your System object definition, the sample time is inherited.

sts = createSampleTime(obj,'Type',Type) creates a sample time specification object with the specified sample time type.

sts = createSampleTime(obj,'Type',Type,Name,Value) creates a sample time specification object with additional options specified by one or more Name,Value pair arguments. Name is a property name and Value is the corresponding value. Name must appear inside quotes. You can specify several name-value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

Dependencies

You can only call createSampleTime from the getSampleTimeImpl method of your System object.
Input Arguments

**obj — System object**
System object

System object that you want to specify the sample time.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'Type','Fixed In Minor Step'`

**Type — Sample time type**

- `'Inherited'`(default) | `'Controllable'` | `'Discrete'` | `'Fixed In Minor Step'`

Type of sample time you want the System object to use in Simulink. For descriptions of the different types of sample times, see:

- `'Inherited'`- “Inherited Sample Time”
- `'Controllable'`- “Controllable Sample Time”
- `'Discrete'`- “Discrete Sample Time”
- `'Fixed In Minor Step'`- “Fixed-in-Minor-Step”

Example: `createSampleTime('Type','Fixed In Minor Step')`

**AllowPropagation — Allow the propagated sample time**

- `'Controllable'` | `'Constant'` | `'Continuous'` | cell array of character vectors | string array

For **Inherited** sample time only.

Specify one or more sample times you want to allow your object to use. When the sample time type is set to **Inherited** and the MATLAB System block receives the specified sample time, the block allows the sample time to propagate. You can specify multiple sample time types as a cell array or string array.
Example: createSampleTime('Type','Inherited','AllowPropagation',
{'Controllable','Continuous'})

AlternatePropagation — Alter the propagated sample time
'Controllable' | 'Constant' | 'Continuous' | cell array of character vectors | string array

For Inherited sample time only.

Specify one or more sample times you do not want your object to use. When the sample
time type is set to Inherited and you do not want the MATLAB System block to use the
specified inherited sample time type, an alternative sample time is used by the block
instead. You can specify multiple sample time types. For each sample time type, Simulink
substitutes a different sample time type:

- 'Controllable' — The base rate of the controlled sample is propagated instead.
  Use this option if your System object depends on having constant time between each
  sample-time hit.
- 'Constant' — The base rate of the model is propagated instead, either the fastest
discrete rate or fixed in minor step.
- 'Continuous' — The fastest discrete rate is propagated instead.

Example: createSampleTime('Type','Inherited','AlternatePropagation','Controllable')

ErrorOnPropagation — Error on the propagated sample time
'Controllable' | 'Constant' | 'Continuous' | cell array of character vectors | string array

For Inherited sample time only.

Specify one or more sample times you do not want your object to use. When the sample
time type is set to Inherited and the MATLAB System block receives the specified
sample time, Simulink throws an error.

Example: createSampleTime('Type','Inherited','ErrorOnPropagation','Controllable')

SampleTime — Time between samples
1 (default) | positive number
For Discrete sample time only.

Specify the time between sample hits in Simulink.

Example: `createSampleTime('Type','Discrete','SampleTime',1)`

Data Types: `single` | `double`

**OffsetTime — Offset from sample time**

0 (default) | nonnegative number less than `SampleTime`

For Discrete sample time only.

Specify the offset time for the sample hit. The offset is a time interval indicating an update delay. The block is updated later in the sample interval than other blocks operating at the same sample rate.

The offset time must be nonnegative and less than `SampleTime`.

Example:
`createSampleTime('Type','Discrete','SampleTime',2,'OffsetTime',1)`

Data Types: `single` | `double`

**TickTime — Time between sample time hits**

-1 (default) | positive scalar

Required for Controllable sample time only.

Specify the time between controllable sample time hits. The tick time must be a positive scalar.

Example:
`createSampleTime('Type','Controllable','TickTime',obj.TickTime)`

Data Types: `single` | `double`

### Output Arguments

**sts — Sample time specification object**

Sample time specification object

The sample time specification object. This object has the following properties:
• Type — Type of sample time
• SampleTime — Time between samples
• OffsetTime — Offset from sample time

Use this object as the return value of `getSampleTimeImpl`.

**Examples**

**Create Inherited Sample Time Specification Object**

Specify that the MATLAB System block inherits the sample from upstream blocks. Inherited sample time is the default, so no additional arguments are needed.

```matlab
function sts = getSampleTimeImpl(obj)
    sts = createSampleTime(obj);
end
```

**Create Discrete Sample Time Object**

Specify a discrete sample time specification for the MATLAB System block, including offset time and the sample time.

```matlab
function sts = getSampleTimeImpl(obj)
    sts = createSampleTime(obj,'Type','Discrete',
                            'SampleTime',10.2,'OffsetTime',0.5);
end
```

**See Also**

`getCurrentTime` | `getSampleTime` | `getSampleTimeImpl` | `setNumTicksUntilNextHit`

**Topics**

“Specify Sample Time for MATLAB System Block System Objects”

**Introduced in R2017b**
**setNumTicksUntilNextHit**

Set the number of ticks in Simulink sample time

**Syntax**

```matlab
setNumTicksUntilNextHit(obj,ticks)
```

**Description**

`setNumTicksUntilNextHit(obj,ticks)` sets the number of ticks in Simulink sample time to wait until the next call to `stepImpl`, or `outputImpl/updateImpl`.

**Dependencies**

To use this method, set your System object to controllable sample time with `createSampleTime('Type','Controllable')`. Otherwise, your System object gives a compilation error.

You can only call this method from `stepImpl`, `outputImpl`, `updateImpl`, or `resetImpl`.

**Input Arguments**

- **obj** — System object
  System object
  System object that you want to specify the sample time.

- **ticks** — Number of ticks in Simulink sample time
  positive integer scalar
  Number of ticks in Simulink sample time to wait until the next call to `stepImpl` or `outputImpl/updateImpl`. Specify this number as a positive integer scalar less than `intmax('uint64')`. 
The number of sample time ticks to wait until the next hit is persistent. If you don’t update this number, Simulink uses the previously set value of number of ticks to wait.

**See Also**
createSampleTime | getCurrentTime | getSampleTime | getSampleTimeImpl

**Topics**
“Specify Sample Time for MATLAB System Block System Objects”

**Introduced in R2018a**
ModelAdvisor.Preferences class

**Package:** ModelAdvisor

Set Model Advisor window preferences by specifying which folders and tabs to display.

**Description**

Use instances of this class to set Model Advisor preferences.

**Construction**

The constructor `ModelAdvisor.Preferences` creates an instance of this class with
default property values.

Create an instance `modelPreferences` of the `ModelAdvisor.Preferences` class.

```plaintext
modelPreferences = ModelAdvisor.Preferences;
```

**Properties**

- **DeselectByProduct — Deselect the By Product folder**
  (default) | true

  Selection of the By Product folder in the Model Advisor window. The default value is true.

  Example: true

  Data Types: logical

- **ShowAccordion — Display advisors**
  (default) | true
Display of the Code Generation Advisor, Upgrade Advisor, and Performance Advisor in the Model Advisor window. You can use these advisors to help configure your model for code generation, upgrade your model for the current release, or improve performance.

Example: true
Data Types: logical

**ShowByProduct — Display the By Product folder**
(default) | true

Display of the By Product folder in the Model Advisor window. The default value is true.

Example: true
Data Types: logical

**ShowByTask — Display the By Task folder**
(default) | true

Display of the By Task folder in the Model Advisor window. The default value is true.

Example: true
Data Types: logical

**ShowExclusionsInRpt — Include exclusions in report**
(default) | true

Include exclusions in the Model Advisor report. The default value is true.

Example: true
Data Types: logical

**ShowExclusionTab — Display the Exclusions tab**
(default) | false

Display of the Exclusions tab in the Model Advisor window. The default value is false. When you click the Exclusions tab, the Model Advisor window displays checks that are excluded form the Model Advisor analysis.

Example: true
Data Types: logical
ShowSourceTab — Display the Source tab
(default) | false

Display of the Source tab in the Model Advisor window. The default value is false. When you click the Source tab, the Model Advisor window displays the check Title, TitleID, and location of the MATLAB source code for the check.

Example: true
Data Types: logical

Examples

Turn Off Display Of By Product Folder

This example shows how to not display the By Product folder in the Model Advisor window:

```matlab
mp = ModelAdvisor.Preferences;
mp.load;
mp.ShowByProduct = false;
mp.save
```

Alternatives

You can set the Model Advisor preferences by using the Model Advisor Preferences dialog box:

- On the Model Advisor menu, select Settings > Preferences.
- From the Model Editor, from the Modeling tab, select Model Advisor > Preferences.

See Also

“Run Model Advisor Checks”

Introduced in R2014b
Simulink.AliasType

Create alias for signal and parameter data type

Description

Use a Simulink.AliasType to create an alias of a built-in data type such as int8.

The name of the object is the alias. The data type to which an alias refers, such as int8, is the base type.

You create the object in the base workspace or a data dictionary. To use the alias, you use the name of the object to set data types for signals, states, and parameters in a model.

Using aliases to specify signal and parameter data types can greatly simplify global changes to the data types that a model specifies. In particular, changing the data type of all signals, states, and parameters whose data type is specified by an alias requires changing only the base type of the alias. By contrast, changing the data types of signals, states, and parameters whose data types are specified by an actual type name requires respecifying the data type of each signal and parameter individually.

You can use objects of this class to create an alias for Simulink built-in data types, fixed-point data types, enumerated data types, Simulink.NumericType objects, and other Simulink.AliasType objects. The code that you generate from a model (Simulink Coder) uses the alias only if you use an ERT-based system target file (Embedded Coder).

Alternatively, to define and name a numeric data type, you can use an object of the class Simulink.NumericType.

Creation

You can use either the Model Explorer or MATLAB commands to create a data type alias.

To use the Model Explorer to create an alias:

You must create data type aliases in the MATLAB workspace or in a data dictionary. If you attempt to create an alias in a model workspace, Simulink software displays an error.

2 From the Model Explorer Add menu, select `Simulink.AliasType`.

Simulink software creates an instance of a `Simulink.AliasType` object and assigns it to a variable named `Alias` in the MATLAB workspace.

3 Rename the variable to a more appropriate name, for example, a name that reflects its intended usage.

To change the name, edit the name displayed in the Name field on the Model Explorer Contents pane.

4 On the Model Explorer Dialog pane, in the Base type field, enter the name of the data type that this alias represents.

You can specify the name of any existing standard or user-defined data type in this field. Skip this step if the base type is `double` (the default).

To generate `Simulink.AliasType` objects that correspond to `typedef` statements in your external C code, consider using the `Simulink.importExternalCTypes` function.

To create a data type alias programmatically, use the `Simulink.AliasType` function described below.

### Syntax

```matlab
aliasObj = Simulink.AliasType
aliasObj = Simulink.AliasType(baseType)
```

### Description

`aliasObj = Simulink.AliasType` returns a `Simulink.AliasType` object with default property values.

`aliasObj = Simulink.AliasType(baseType)` returns a `Simulink.AliasType` object and initializes the value of the BaseType property by using `baseType`. 
Properties

For information about properties in the property dialog box of a Simulink.AliasType object, see “Simulink.AliasType Property Dialog Box”.

**BaseType — Name of base data type**

'**double**' (default) | character vector

Name of the base data type that this alias renames, specified as a character vector. You can specify the name of a standard data type, such as `int8`, or the name of a custom data type, such as the name of another Simulink.AliasType object or the name of an enumeration.

To specify a fixed-point data type, you can use a call to the `fixdt` function, such as `fixdt(0,16,7)`.

You can, with one exception, specify a nonstandard data type, e.g., a data type defined by a Simulink.NumericType object, by specifying the data type name. The exception is a Simulink.NumericType whose `DataTypeMode` is `Fixed-point: unspecified scaling`.

**Note** Fixed-point: unspecified scaling is a partially specified type whose definition is completed by the block that uses the Simulink.NumericType. Forbidding its use in alias types avoids creating aliases that have different base types depending on where they are used.

Corresponds to **Base type** in the property dialog box.

Example: 'int8'

Example: 'myOtherAlias'

Data Types: char

**DataScope — Specification to generate or import type definition in the generated code**

'**Auto**' (default) | 'Exported' | 'Imported'

Specification to generate or import the type definition (typedef) in the generated code (Simulink Coder), specified as 'Auto', 'Exported', or 'Imported'.
The table shows the effect of each option.

<table>
<thead>
<tr>
<th>Value</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>'Auto'</td>
<td>If no value is specified for HeaderFile, export the type definition to <code>model_types.h</code>, where <code>model</code> is the model name. If you have an Embedded Coder license, and you have specified a data type replacement, then export the type definition to <code>rtwtypes.h</code>. If a value is specified for HeaderFile, import the data type definition from the specified header file.</td>
</tr>
<tr>
<td>'Exported'</td>
<td>Export the data type definition to a header file, which can be specified in the HeaderFile property. If no value is specified for HeaderFile, the header file name defaults to <code>type.h</code>. <code>type</code> is the data type name.</td>
</tr>
<tr>
<td>'Imported'</td>
<td>Import the data type definition from a header file, which can be specified in the HeaderFile property. If no value is specified for HeaderFile, the header file name defaults to <code>type.h</code>. <code>type</code> is the data type name.</td>
</tr>
</tbody>
</table>

For more information, see “Control File Placement of Custom Data Types” (Embedded Coder).

Corresponds to **Data scope** in the property dialog box.

**Description — Custom description of data type alias**

'' (empty character vector) (default) | character vector

Custom description of the data type alias, specified as a character vector.

Corresponds to **Description** in the property dialog box.

Example: 'This type alias corresponds to a floating-point implementation.'

Data Types: char

**HeaderFile — Name of header file that contains type definition in the generated code**

'' (empty character vector) (default) | character vector

Name of the header file that contains the type definition (typedef) in the generated code, specified as a character vector.
If this property is specified, the specified name is used during code generation for importing or exporting. If this property is empty, the value defaults to `type.h` if `DataScope` equals 'Imported' or 'Exported', or defaults to `model_types.h` if `DataScope` equals 'Auto'.

By default, the generated `#include` directive uses the preprocessor delimiter " instead of `< and >. To generate the directive `#include <myTypes.h>`, specify `HeaderFile` as '<myTypes.h>'.

For more information, see “Control File Placement of Custom Data Types” (Embedded Coder).

Corresponds to **Header file** in the property dialog box.

Example: 'myHdr.h'
Example: 'myHdr'
Example: 'myHdr.hpp'
Data Types: char

### Examples

**Create Alias for Enumerated Data Type**

To create an alias for an enumerated type called `SlDemoSign`:

```
myEnumAlias = Simulink.AliasType('Enum: SlDemoSign');
```

**Create Alias for Fixed-Point Data Type**

To create an alias for a fixed-point data type by using a `Simulink.AliasType` object, set the `BaseType` property of the object by using a call to the `fixdt` function. The value of `BaseType` must be specified as a character vector.

For example, this code creates an alias for an unsigned fixed-point data type with word length 16 and fraction length 7.
myFixptAlias = Simulink.AliasType;
myFixptAlias.BaseType = 'fixdt(0,16,7)';

See Also
Simulink.NumericType

Topics
“Control Signal Data Types”
“Control Data Type Names in Generated Code” (Embedded Coder)
“About Data Types in Simulink”

Introduced before R2006a
Simulink.Annotation

Specify properties of model annotation

Description

Instances of this class specify the properties of annotations. You can use getCallbackAnnotation in an annotation callback function to get the Simulink.Annotation instance for the annotation associated with the callback function. You can use find_system and get_param to get the Simulink.Annotation instance associated with any annotation in a model. For example, this code gets the annotation object for the first annotation in the currently selected model and turns on its drop shadow

```matlab
ah = find_system(gcs,'FindAll','on','type','annotation');
ao = get_param(ah(1),'Object');
ao.DropShadow = 'on';
```

Children

None.

Property Summary

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>AnnotationType</td>
<td>Type of annotation—text (note), area, or image. This property is read only.</td>
<td>note_annotation</td>
</tr>
<tr>
<td>Property</td>
<td>Description</td>
<td>Values</td>
</tr>
<tr>
<td>------------------</td>
<td>--------------------------------------------------</td>
<td>------------------------------------------------------------------------</td>
</tr>
<tr>
<td>BackgroundColor</td>
<td>Background color of this annotation.</td>
<td>RGB value array $[r,g,b,a]$ where $r$, $g$, $b$, and $a$ are the red, green, blue, and alpha values of the color normalized to the range $0.0$ to $1.0$, delineated with commas. The alpha value is optional and ignored. Annotation background color can also be 'automatic' (transparent), 'black', 'white', 'red', 'green', 'blue', 'cyan', 'magenta', 'yellow', 'gray', 'lightBlue', 'orange', 'darkGreen'.</td>
</tr>
<tr>
<td>ClickFcn</td>
<td>Specifies MATLAB code to execute when you click this annotation. See “Associate a Click Function with an Annotation” for more information.</td>
<td>character vector</td>
</tr>
<tr>
<td>DeleteFcn</td>
<td>MATLAB code to execute before deleting this annotation. See “Annotation Callback Functions”.</td>
<td>character vector</td>
</tr>
<tr>
<td>Description</td>
<td>Description of this annotation.</td>
<td>character vector</td>
</tr>
<tr>
<td>DropShadow</td>
<td>Turn drop shadow display on or off.</td>
<td>'on'</td>
</tr>
<tr>
<td>FixedHeight</td>
<td>Specify whether the bottom border of the annotation resizes as you add content.</td>
<td>'on'</td>
</tr>
<tr>
<td>FixedWidth</td>
<td>Specify whether to use word wrap or to have the width of the annotation expand to accommodate text.</td>
<td>'on'</td>
</tr>
<tr>
<td>Property</td>
<td>Description</td>
<td>Values</td>
</tr>
<tr>
<td>------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------------------------------------------------------------------------</td>
</tr>
<tr>
<td>FontAngle (plain text only)</td>
<td>Angle of the annotation font. The default value, 'auto', uses of the default font angle specified for lines in the Font Styles dialog box.</td>
<td>'normal'</td>
</tr>
<tr>
<td>FontName (plain text only)</td>
<td>Name of annotation font. The default value, 'auto', uses the default font specified for lines in the Font Styles dialog box.</td>
<td>character vector</td>
</tr>
<tr>
<td>FontSize (plain text only)</td>
<td>Size of annotation font in points. The default value, -1, uses the default text size for lines specified in the Font Styles dialog box.</td>
<td>decimal number</td>
</tr>
<tr>
<td>FontWeight (plain text only)</td>
<td>Weight of the annotation font. The default value, 'auto', use of the default weight for lines specified in the Font Styles dialog box.</td>
<td>'light'</td>
</tr>
<tr>
<td>ForegroundColor</td>
<td>Foreground color of this annotation.</td>
<td>RGB value array</td>
</tr>
<tr>
<td>Annotation foreground color can also be 'black', 'white', 'red', 'green', 'blue', 'cyan', 'magenta', 'yellow', 'gray', 'lightBlue', 'orange', 'darkGreen'.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Handle</td>
<td>Annotation handle.</td>
<td>real</td>
</tr>
<tr>
<td>HiliteAncestors</td>
<td>For internal use.</td>
<td></td>
</tr>
<tr>
<td>Horizontal-Alignment</td>
<td>Horizontal alignment of this annotation.</td>
<td>'center'</td>
</tr>
<tr>
<td>Property</td>
<td>Description</td>
<td>Values</td>
</tr>
<tr>
<td>-------------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>-------------------------</td>
</tr>
<tr>
<td>Interpreter</td>
<td>Specifies whether the annotation is interpreted as rich text or contains LaTeX commands. The default value, 'off', specifies a plain text annotation. When you format a plain text annotation using the formatting toolbar, the annotation becomes a rich text annotation.</td>
<td>'rich'</td>
</tr>
<tr>
<td>IsImage</td>
<td>Specifies whether the annotation is an image-only annotation.</td>
<td>'on'</td>
</tr>
<tr>
<td>InternalMargins</td>
<td>Space from the bounding box of text to the borders of the annotation.</td>
<td>1x4 array [left top right bottom]. The default is [1 1 1 1]. Supported coordinates are between -1073740824 and 1073740823, inclusive.</td>
</tr>
<tr>
<td>LoadFcn</td>
<td>MATLAB code to execute when the model containing this annotation is loaded. See “Annotation Callback Functions”.</td>
<td>character vector</td>
</tr>
<tr>
<td>Name</td>
<td>Text of annotation. Same as Text.</td>
<td>character vector</td>
</tr>
<tr>
<td>Parent</td>
<td>Parent name of annotation object.</td>
<td>character vector</td>
</tr>
<tr>
<td>Path</td>
<td>Path to the annotation.</td>
<td>character vector</td>
</tr>
<tr>
<td>PlainText</td>
<td>Read-only display of the text in the annotation, without formatting.</td>
<td>vector</td>
</tr>
<tr>
<td>Property</td>
<td>Description</td>
<td>Values</td>
</tr>
<tr>
<td>-----------------</td>
<td>------------------------------------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>Position</td>
<td>Location of the annotation.</td>
<td>vector of coordinates, in pixels: [left top right bottom] The origin is the upper-left corner of the Simulink Editor canvas before any canvas resizing. Supported coordinates are between -1073740824 and 1073740823, inclusive. Positive values are to the right of and down from the origin. Negative values are to the left of and up from the origin.</td>
</tr>
<tr>
<td>RequirementInfo</td>
<td>For internal use.</td>
<td>character vector</td>
</tr>
<tr>
<td>Selected</td>
<td>Specifies whether this annotation is currently selected.</td>
<td>'on'</td>
</tr>
<tr>
<td>Tag</td>
<td>Text to assign to the annotation Tag parameter and save with the annotation.</td>
<td>character vector</td>
</tr>
<tr>
<td>TeXMode</td>
<td>Specifies whether to render TeX markup.</td>
<td>'on'</td>
</tr>
<tr>
<td>Text</td>
<td>Text of annotation. Same as Name.</td>
<td>character vector</td>
</tr>
<tr>
<td>Type</td>
<td>Annotation type. This is always 'annotation'.</td>
<td>'annotation'</td>
</tr>
<tr>
<td>Property</td>
<td>Description</td>
<td>Values</td>
</tr>
<tr>
<td>----------------------------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>-------------------------</td>
</tr>
<tr>
<td>UseDisplayText-AsClickCallback</td>
<td>Specifies whether to use the contents of the Text property as the click function for this annotation. If set to 'on', the text of the annotation is interpreted as a valid MATLAB expression and run. If set to 'off', clicking the annotation runs the click function, if there is one. If there is no click function, clicking the annotation has no effect. See “Associate a Click Function with an Annotation” for more information.</td>
<td>'on'</td>
</tr>
<tr>
<td>UserData</td>
<td>Any data that you want to associate with this annotation.</td>
<td>vector</td>
</tr>
<tr>
<td>VerticalAlignment</td>
<td>Vertical alignment of this annotation.</td>
<td>'middle'</td>
</tr>
</tbody>
</table>

### Method Summary

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>delete</td>
<td>Delete this annotation from the Simulink model.</td>
</tr>
<tr>
<td>dialog</td>
<td>Display the Annotation properties dialog box.</td>
</tr>
<tr>
<td>disp</td>
<td>Display the property names and their settings for this Annotation object.</td>
</tr>
<tr>
<td>fitToView</td>
<td>Zoom in on this annotation and highlight it in the model.</td>
</tr>
<tr>
<td>get</td>
<td>Return the specified property settings for this annotation.</td>
</tr>
<tr>
<td>help</td>
<td>Display a list of properties for this Annotation object with short descriptions.</td>
</tr>
<tr>
<td>methods</td>
<td>Display all nonglobal methods of this Annotation object.</td>
</tr>
<tr>
<td>Method</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>set</td>
<td>Set the specified property of this Annotation object with the specified value.</td>
</tr>
<tr>
<td>setImage</td>
<td>Set the annotation contents to the specified image file. The resulting annotation is an image-only annotation.</td>
</tr>
<tr>
<td>struct</td>
<td>Return and display a MATLAB structure containing the property settings of this Annotation object.</td>
</tr>
<tr>
<td>view</td>
<td>Display this annotation in the Simulink Editor with this annotation highlighted.</td>
</tr>
</tbody>
</table>

**Related Links**

“Create an Annotation Programmatically”

**Introduced before R2006a**
**Simulink.BlockCompDworkData**

Provide postcompilation information about block's DWork vector

**Description**

Simulink software returns an instance of this class when a MATLAB program, e.g., a Level-2 MATLAB S-function, invokes the “Dwork” on page 5-510 method of a block's runtime object after the model containing the block has been compiled.

**Parent**

Simulink.BlockData

**Children**

None

**Property Summary**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Usage” on page 5-215</td>
<td>Usage type of this DWork vector.</td>
</tr>
<tr>
<td>“UsedAsDiscState” on page 5-215</td>
<td>True if this DWork vector is being used to store the values of a block's discrete states.</td>
</tr>
</tbody>
</table>
Properties

Usage

Description

Returns a character vector indicating how this DWork vector is used. Permissible values are:

- DWork
- DState
- Scratch
- Mode

Data Type

character vector

Access

RW for MATLAB S-function blocks, RO for other blocks.

UsedAsDiscState

Description

True if this DWork vector is being used to store the values of a block's discrete states.

Data Type

Boolean

Access

RW for MATLAB S-Function blocks, RO for other blocks.

Introduced before R2006a
**Simulink.BlockCompInputPortData**

Provide postcompilation information about block input port

**Description**

Simulink software returns an instance of this class when a MATLAB program, e.g., a Level-2 MATLAB S-function, invokes the “InputPort” on page 5-512 method of a block's run-time object after the model containing the block has been compiled.

**Parent**

Simulink.BlockPortData

**Children**

None

**Property Summary**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>“DirectFeedthrough” on page 5-217</td>
<td>True if this port has direct feedthrough.</td>
</tr>
<tr>
<td>“Overwritable” on page 5-217</td>
<td>True if this port is overwritable.</td>
</tr>
</tbody>
</table>
Properties

DirectFeedthrough

Description
True if this input port has direct feedthrough.

Data Type
Boolean

Access
RW for MATLAB S functions, R0 for other blocks.

Overwritable

Description
True if this input port is overwritable.

Data Type
Boolean

Access
RW for MATLAB S functions, R0 for other blocks.

Introduced before R2006a
Simulink.BlockCompOutputPortData

Provide postcompilation information about block output port

**Description**

Simulink software returns an instance of this class when a MATLAB program, e.g., a Level-2 MATLAB S-function, invokes the “OutputPort” on page 5-512 method of a block's run-time object after the model containing the block has been compiled.

**Parent**

Simulink.BlockPortData

**Children**

None

**Property Summary**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Reusable” on page 5-242</td>
<td>Specifies whether an output port's memory is reusable.</td>
</tr>
</tbody>
</table>

**Properties**

**Reusable**

**Description**

Specifies whether an output port's memory is reusable. Options are: NotReusableAndGlobal and ReusableAndLocal.
Data Type
character vector

Access
RW for MATLAB S functions, R0 for other blocks.

Introduced before R2006a
Simulink.BlockData

Provide run-time information about block-related data, such as block parameters

Description

This class defines properties that are common to objects that provide run-time information about a block's ports and work vectors.

Parent

None

Children

Simulink.BlockPortData, Simulink.BlockCompDworkData

Property Summary

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>“AliasedThroughDataType” on page 5-221</td>
<td>Fundamental base data type.</td>
</tr>
<tr>
<td>“AliasedThroughDataTypeID” on page 5-222</td>
<td>Fundamental base data type ID.</td>
</tr>
<tr>
<td>“Complexity” on page 5-222</td>
<td>Numeric type (real or complex) of the block data.</td>
</tr>
<tr>
<td>“Data” on page 5-222</td>
<td>The block data.</td>
</tr>
<tr>
<td>“DataAsDouble” on page 5-223</td>
<td>The block data in double form.</td>
</tr>
<tr>
<td>“Datatype” on page 5-223</td>
<td>Data type of the block data.</td>
</tr>
<tr>
<td>“DatatypeID” on page 5-223</td>
<td>Index of the data type of the block data.</td>
</tr>
</tbody>
</table>
### Properties

**AliasedThroughDataType**

**Description**

Data type aliases allow a data type (B) to be recursively aliased to another alias type or `BaseType` (A). If alias type A is aliased to another alias type that is aliased to another alias type and so forth, this property allows the alias type to be iteratively searched (aliased through) until the type is no longer an alias type and that final result is the value of the property returned. For example, assume that you have created the Simulink Alias types A and B as follows:

```matlab
A = Simulink.AliasType('double')

A = Simulink.AliasType('double')
   Description: '
   HeaderFile: '
   BaseType: 'double'
B = Simulink.AliasType('A')

B = Simulink.AliasType('A')
   Description: '
   HeaderFile: '
   BaseType: 'A'
```

If the data type of an item of block data is B, this property returns the base type A instead of B.

**Data Type**

character vector
**Access**
R0

**AliasedThroughDataTypeID**

**Description**
Index of the data type alias returned by the AliasedThroughDataType property.

**Data Type**
integer

**Access**
R0

**Complexity**

**Description**
Numeric type (real or complex) of the block data.

**Data Type**
character vector

**Access**
RW for MATLAB S functions, R0 for other blocks.

**Data**

**Description**
The block data.

**Data Type**
The data type specified by the “Datatype” on page 5-223 or “DatatypeID” on page 5-223 properties of this object.
**Access**

RW

**DataAsDouble**

**Description**

The block data's in double form.

**Data Type**

double

**Access**

R0

**Datatype**

**Description**

Data type of the values of the block-related object.

**Data Type**

character vector

**Access**

R0

**DatatypeID**

**Description**

Index of the data type of the values of the block-related object. Enter the numeric value for the desired data type, as follows:
### Data Type

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>'inherited'</td>
<td>-1</td>
</tr>
<tr>
<td>'double'</td>
<td>0</td>
</tr>
<tr>
<td>'single'</td>
<td>1</td>
</tr>
<tr>
<td>'int8'</td>
<td>2</td>
</tr>
<tr>
<td>'uint8'</td>
<td>3</td>
</tr>
<tr>
<td>'int16'</td>
<td>4</td>
</tr>
<tr>
<td>'uint16'</td>
<td>5</td>
</tr>
<tr>
<td>'int32'</td>
<td>6</td>
</tr>
<tr>
<td>'uint32'</td>
<td>7</td>
</tr>
<tr>
<td>'boolean' or fixed-point data types</td>
<td>8</td>
</tr>
</tbody>
</table>

**Data Type**

integer

**Access**

RW for MATLAB S functions, R0 for other blocks

**Dimensions**

**Description**

Dimensions of the block-related object, e.g., parameter or DWork vector.

**Data Type**

array

**Access**

RW for MATLAB S functions, R0 for other blocks
Name

Description

Name of block-related object, e.g., a block parameter or DWork vector.

Data Type

character vector

Access

RW for MATLAB S functions, RO for other blocks

Type

Description

Type of block data. Possible values are:

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'BlockPreCompInputPortData'</td>
<td>This object contains data for an input port before the model is compiled.</td>
</tr>
<tr>
<td>'BlockPreCompOutputPortData'</td>
<td>This object contains data for an output port before the model is compiled.</td>
</tr>
<tr>
<td>'BlockCompInputPortData'</td>
<td>This object contains data for an input port after the model is compiled.</td>
</tr>
<tr>
<td>'BlockCompOutputPortData'</td>
<td>This object contains data for an output port after the model is compiled.</td>
</tr>
<tr>
<td>'BlockPreCompDworkData'</td>
<td>This object contains data for a DWork vector before the model is compiled.</td>
</tr>
<tr>
<td>'BlockCompDworkData'</td>
<td>This object contains data for a DWork vector after the model is compiled.</td>
</tr>
</tbody>
</table>
| 'BlockDialogPrmData'              | This object describes a dialog box parameter of a Level-2 MATLAB S-function.
<p>| 'BlockRuntimePrmData'             | This object describes a run-time parameter of a Level-2 MATLAB S-function.  |</p>
<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'BlockCompContStatesData'</td>
<td>This object describes the continuous states of the block at the current time step.</td>
</tr>
<tr>
<td>'BlockDerivativesData'</td>
<td>This object describes the derivatives of the block's continuous states at the current time step.</td>
</tr>
</tbody>
</table>

**Data Type**

character vector

**Access**

R0

**Introduced before R2006a**
Simulink.BlockPath

Fully specified Simulink block path

Description

A Simulink.BlockPath object represents a fully specified block path that uniquely identifies a block within a model hierarchy, including model reference hierarchies that involve multiple instances of a referenced model. Simulink uses block path objects in a variety of contexts. For example, when you specify normal mode visibility, Simulink uses block path objects to identify the models with Normal mode visibility. For details, see “Normal Mode Visibility”.

The Simulink.BlockPath class is very similar to the Simulink.SimulationData.BlockPath class.

You must have Simulink installed to use the Simulink.BlockPath class. However, you do not have to have Simulink installed to use the Simulink.SimulationData.BlockPath class. If you have Simulink installed, consider using Simulink.BlockPath instead of Simulink.SimulationData.BlockPath, because the Simulink.BlockPath class includes a method for checking the validity of block path objects without you having to update the model diagram.

Property Summary

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SubPath on page 5-228</td>
<td>Individual component within the block specified by the block path</td>
</tr>
</tbody>
</table>

Method Summary

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BlockPath on page 5-229</td>
<td>Create a block path.</td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>convertToCell on page 5-231</td>
<td>Convert a block path to a cell array of character vectors.</td>
</tr>
<tr>
<td>getBlock on page 5-232</td>
<td>Get a single block path in the model reference hierarchy.</td>
</tr>
<tr>
<td>getLength on page 5-233</td>
<td>Get the length of the block path.</td>
</tr>
<tr>
<td>open on page 5-234</td>
<td>Open the specified model, library, subsystem, or block.</td>
</tr>
<tr>
<td>validate on page 5-235</td>
<td>Determine whether the block path represents a valid block hierarchy.</td>
</tr>
</tbody>
</table>

### Properties

**SubPath**

#### Description

Represents an individual component within the block specified by the block path.

For example, if the block path refers to a Stateflow chart, you can use `SubPath` to indicate the chart signals. For example:

Block Path:

    'sf_car/shift_logic'

    SubPath:

        'gear_state.first'

**Data Type**

character vector

**Access**

RW
Methods

BlockPath

Purpose
Create block path

Syntax

blockpath_object = Simulink.BlockPath()
blockpath_object = Simulink.BlockPath(blockpath)
blockpath_object = Simulink.BlockPath(paths)
blockpath_object = Simulink.BlockPath(paths, subpath)

Input Arguments

blockpath
Block path object that you want to copy.

paths
A character vector or cell array of character vectors that Simulink uses to build the block path.

Specify each character vector in order, from the top model to the specific block for which you are creating a block path.

Each character vector must be a path to a block within the Simulink model. The block must be:

• A block in a single model
• A Model block (except for the last character vector, which may be a block other than a Model block)
• A block that is in a model that is referenced by a Model block that is specified in the previous character vector

When you create a block path for specifying Normal mode visibility:

• The first character vector must represent a block that is in the top model in the model reference hierarchy.
• Character vectors must represent Model blocks that are in Normal mode.
• Character vectors that represent variant models or variant subsystems must refer to an active variant.

You can use `gcb` in the cell array to specify the currently selected block.

`subpath`  
Character vector that represents an individual component within a block.

**Output Arguments**

`blockpath_object`  
Block path that you create.

**Description**

`blockpath_object = Simulink.BlockPath()` creates an empty block path.

`blockpath_object = Simulink.BlockPath(blockpath)` creates a copy of the block path of the block path object that you specify with the `source_blockpath` argument.

`blockpath = Simulink.BlockPath(paths)` creates a block path from the cell array of character vectors that you specify with the `paths` argument. Each character vector represents a path at a level of model hierarchy. Simulink builds the full block path based on the character vectors.

`blockpath = Simulink.BlockPath(paths, subpath)` creates a block path from the character vector or cell array of character vectors that you specify with the `paths` argument and creates a path for the individual component (for example, a signal) of the block.

**Example**

Create a block path object called `bp1`, using `gcb` to get the current block.

```matlab
sldemo_mdlref_depgraph
bp1 = Simulink.BlockPath(gcb)
```

The resulting block path is the top-level Model block called `thermostat` (the top-left Model block).

`bp1 =`
Create a block path object called `bp2`, using a cell array of character vectors representing elements of the block path.

```matlab
sldemo_mdlref_depgraph
bp2 = Simulink.BlockPath({'sldemo_mdlref_depgraph/thermostat', ...
'sldemo_mdlref_heater/Fahrenheit to Celsius', ...
'sldemo_mdlref_F2C/Gain1'})
```

The resulting block path reflects the model reference hierarchy for the block path

```matlab
bp2 =

Simulink.BlockPath
Package: Simulink

Block Path:
'sldemo_mdlref_depgraph/thermostat'
   'sldemo_mdlref_heater/Fahrenheit to Celsius'
   'sldemo_mdlref_F2C/Gain1'
```

### convertToCell

**Purpose**

Convert block path to cell array of character vectors

**Syntax**

```matlab
cellarray = Simulink.BlockPath.convertToCell()
```

**Output Arguments**

`cellarray`

Cell array of character vectors representing elements of block path.
**Description**

cellarray = Simulink.BlockPath.convertToCell() converts a block path to a cell array of character vectors.

**Examples**

```matlab
sldemo_mdlref_depgraph
bp2 = Simulink.BlockPath({'sldemo_mdlref_depgraph/thermostat', ...
'sldemo_mdlref_heater/Fahrenheit to Celsius', ...
'sldemo_mdlref_F2C/Gain1'})
cellarray_for_bp2 = bp2.convertToCell()
```

The result is a cell array representing the elements of the block path.

```matlab
cellarray_for_bp2 =
    'sldemo_mdlref_depgraph/thermostat'
    'sldemo_mdlref_heater/Fahrenheit to Celsius'
    'sldemo_mdlref_F2C/Gain1'
```

**getBlock**

**Purpose**

Get block path in model reference hierarchy

**Syntax**

```matlab
block = Simulink.BlockPath.getBlock(index)
```

**Input Arguments**

```matlab
index
```

The index of the block for which you want to get the block path. The index reflects the level in the model reference hierarchy. An index of 1 represents a block in the top-level model, an index of 2 represents a block in a model referenced by the block of index 1, and an index of n represents a block that the block with index n - 1 references.
**Output Arguments**

block

The block representing the level in the model reference hierarchy specified by the index argument.

**Description**

blockpath = Simulink.BlockPath.getBlock(index) returns the block path of the block specified by the index argument.

**Example**

Get the block for the second level in the model reference hierarchy.

```matlab
sldemo_mdlref_depgraph
bp2 = Simulink.BlockPath({'sldemo_mdlref_depgraph/thermostat', ...
'sldemo_mdlref_heater/Fahrenheit to Celsius', ...
'sldemo_mdlref_F2C/Gain1'})
blockpath = bp2.getBlock(2)
```

The result is the thermostat block, which is at the second level in the block path hierarchy.

```matlab
blockpath =
```

```matlab
sldemo_mdlref_heater/Fahrenheit to Celsius
```

**getLength**

**Purpose**

Get length of block path

**Syntax**

```matlab
length = Simulink.BlockPath.getLength()
```

**Output Arguments**

length

The length of the block path. The length is the number of levels in the model reference hierarchy.
Description

length = Simulink.BlockPath.getLength() returns a numeric value that corresponds to the number of levels in the model reference hierarchy for the block path.

Example

Get the length of block path bp2.

sldemo_mdlref_depgraph
bp2 = Simulink.BlockPath({'sldemo_mdlref_depgraph/thermostat', ...
'sldemo_mdlref_heater/Fahrenheit to Celsius', ...
'sldemo_mdlref_F2C/Gain1'})
length_bp2 = bp2.getLength()

The result reflects that the block path has three elements.

length_bp2 =

3

open

Purpose

Open the specified model, library, subsystem, or block.

Syntax

Simulink.BlockPath.open()
Simulink.BlockPath.open('OpenType', openType)
Simulink.BlockPath.open('Force', forceOpen)

Input Arguments

openType

Specify where to open a subsystem or model specified by a Simulink.BlockPath object. Options include:

- 'REUSE_TAB' (default) — Open in the current window tab.
- 'NEW_TAB' — Open in a new tab.
- 'NEW_WINDOW' — Open in a new window.
forceOpen

Force the Simulink.BlockPath object to open, regardless of block mask. Options include:

- 'off' (default) — opens the block mask.
- 'on' — opens the dialog box of the block under the mask or opens a masked system in a new tab.

Description

Simulink.BlockPath.open() opens the specified model, library, subsystem, or block. This is equivalent to double-clicking the model or library in the Current Folder Browser, or the subsystem or block in the Simulink Editor.

Simulink.BlockPath.open('OpenType', openType) allows you to specify whether a subsystem or model opens in a new tab or window.

Simulink.BlockPath.open('Force', forceOpen) looks under the mask of a masked block. It opens the dialog box of the block under the mask or opens a masked system in a new tab. This option is equivalent to the Look Under Mask menu item.

Example

Open model sldemo_mdlref_depgraph.

sldemo_mdlref_depgraph

Select a block in the model hierarchy.

bp3 = Simulink.BlockPath({'sldemo_mdlref_depgraph/thermostat', 'sldemo_mdlref_heater/Fahrenheit to Celsius'})

Open the referenced model specified by bp3 in a new tab.

bp3.open('OpenType','NEW_TAB')

validate

Purpose

Determine whether block path represents valid block hierarchy
Syntax

Simulink.BlockPath.validate()
Simulink.BlockPath.validate(AllowInactiveVariant)

Description

Simulink.BlockPath.validate() determines whether the block path represents a valid block hierarchy. If there are any validity issues, messages appear in the MATLAB command window. The method checks that:

- All elements in the block path represent valid blocks.
- Each element except for the last element:
  - Is a valid Model block
  - References the model of the next element

See Also

Simulink.BlockPortData

Describe block input or output port

Description

This class defines properties that are common to objects that provide run-time information about a block's ports.

Parent

Simulink.BlockData

Children

Simulink.BlockPreCompInputPortData, Simulink.BlockPreCompOutputPortData, Simulink.BlockCompInputPortData, Simulink.BlockCompOutputPortData

Property Summary

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>“IsBus” on page 5-238</td>
<td>True if this port is connected to a bus.</td>
</tr>
<tr>
<td>“IsSampleHit” on page 5-238</td>
<td>True if this port produces output or accepts input at the current simulation time step.</td>
</tr>
<tr>
<td>“SampleTime” on page 5-238</td>
<td>Sample time of this port.</td>
</tr>
<tr>
<td>“SampleTimeIndex” on page 5-239</td>
<td>Sample time index of this port.</td>
</tr>
</tbody>
</table>
Properties

IsBus

Description
True if this port is connected to a bus.

Data Type
Boolean

Access
RO

IsSampleHit

Description
True if this port produces output or accepts input at the current simulation time step.

Data Type
Boolean

Access
RO

SampleTime

Description
Sample time of this port.

Data Type

[period offset] where period and offset are values of type double. See “Specify Sample Time” for more information.
**Access**

RW for MATLAB S functions, R0 for other blocks

**SampleTimeIndex**

**Description**

Sample time index of this port.

**Data Type**

integer

**Access**

R0

**Introduced before R2006a**
**Simulink.BlockPreCompInputPortData**

Provide precompilation information about block input port

**Description**

Simulink software returns an instance of this class when a MATLAB program, e.g., a Level-2 MATLAB S-function, invokes the “InputPort” on page 5-512 method of a block's run-time object before the model containing the block has been compiled.

**Parent**

Simulink.BlockPortData

**Children**

None

**Property Summary**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>“DirectFeedthrough” on page 5-241</td>
<td>True if this port has direct feedthrough.</td>
</tr>
<tr>
<td>“Overwritable” on page 5-241</td>
<td>True if this port is overwritable.</td>
</tr>
</tbody>
</table>
Properties

DirectFeedthrough

Description
True if this input port has direct feedthrough.

Data Type
Boolean

Access
RW for MATLAB S functions, R0 for other blocks

Overwritable

Description
True if this input port is overwritable.

Data Type
Boolean

Access
RW for MATLAB S functions, R0 for other blocks

Introduced before R2006a
Simulink.BlockPreCompOutputPortData

Provide precompilation information about block output port

Description

Simulink software returns an instance of this class when a MATLAB program, e.g., a Level-2 MATLAB S-function, invokes the “OutputPort” on page 5-512 method of a block's run-time object before the model containing the block has been compiled.

Parent

Simulink.BlockPortData

Children

none

Property Summary

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Reusable” on page 5-242</td>
<td>Specifies whether an output port's memory is reusable.</td>
</tr>
</tbody>
</table>

Properties

Reusable

Description

Specifies whether an output port's memory is reusable. Options are: NotReusableAndGlobal and ReusableAndLocal.
**Data Type**

character vector

**Access**

RW for MATLAB S functions, RO for other blocks

**Introduced before R2006a**
Simulink.Breakpoint class

**Package:** Simulink

Store and share data for a breakpoint set, configure the data for ASAP2 and AUTOSAR code generation

**Description**

An object of the Simulink.Breakpoint class stores breakpoint set data for a lookup table. You can use that data in one or more Prelookup blocks. With the object, you can specify a data type and code generation settings for the breakpoint set and share the set between multiple lookup tables. Use Simulink.Breakpoint objects and Simulink.LookupTable objects to configure COM_AXIS code generation for calibration.

The code generated for a Simulink.Breakpoint object is an array or a structure with two fields. If you configure the object to appear as a structure, one field stores the specified breakpoint set data and one scalar field stores the number of elements in the breakpoint set data. You can configure the structure type name, the field name, and other characteristics by using the properties of the object.

To package lookup table and breakpoint set data into a single structure in the generated code, for example, for STD_AXIS code generation, use a Simulink.LookupTable object to store all of the data. See “Package Shared Breakpoint and Table Data for Lookup Tables”.

**Construction**


To create a Simulink.Breakpoint object by using the Model Explorer, use the button on the toolbar. The default name for the object is Object.
Property Dialog Box

**Breakpoints**

The breakpoint set information. You can configure these characteristics:

**Support tunable size**

Specification to enable tuning the effective size of the table in the generated code. If you select this option, in the generated code, the `Simulink.Breakpoint` object appears as a structure variable. The structure has one field to store the breakpoint vector data and one field to store the number of elements in the breakpoint vector. You can change the value of the second field to adjust the effective size of the table.

If you clear this option, the `Simulink.Breakpoint` object appears in the generated code as a separate array variable instead of a structure.

**Value**

Breakpoint set data. Specify a vector with at least two elements.

You can also use an expression with mathematical operators such as `sin(1:0.5:30)` as long as the expression returns a numeric vector. When you click **Apply** or **OK**, the object executes the expression and uses the result to set the value of this property.
When you set **Data type** to **auto**, to set **Value**, use a typed expression such as single([1 2 3]) or use the **fi** constructor to embed an **fi** object.

You can edit this data by using a more intuitive interface in a lookup table block. See “Import Lookup Table Data from MATLAB”.

**Data type**

Data type of the breakpoint set. The default setting is **auto**, which means that the breakpoint set acquires a data type from the value that you specify in **Value**. If you use an untyped expression such as [1 2 3] to set **Value**, the breakpoint data use the data type **double**. If you specify a typed expression such as single([1 2 3]) or an **fi** object, the breakpoint data use the data type specified by the expression or object. Enumerated data types are also supported.

You can explicitly specify an integer data type, a floating-point data type, a fixed-point data type, or a data type expression such as the name of a Simulink.**AliasType** object.

For more information about data types in Simulink, see “Data Types Supported by Simulink”. To decide how to control the data types of table and breakpoint data in Simulink.**LookupTable** and Simulink.**Breakpoint** objects, see “Control Data Types of Lookup Table Objects” (Simulink Coder).

**Dimensions**

Dimension lengths of the breakpoint set.

To use symbolic dimensions, specify a character vector. See “Implement Dimension Variants for Array Sizes in Generated Code” (Embedded Coder).

**Min**

Minimum value of the elements in the breakpoint set. The default value is empty, [ ]. You can specify a numeric, real value.

For more information about how Simulink uses this property, see “Specify Minimum and Maximum Values for Block Parameters”.

**Max**

Maximum value of the elements in the breakpoint set. The default value is empty, [ ]. You can specify a numeric, real value.

For more information about how Simulink uses this property, see “Specify Minimum and Maximum Values for Block Parameters”.
**Stored Int Min**

For Simulink.Breakpoint objects with a fixed-point data type, the minimum value of the elements in the breakpoint set, specified as a stored integer value. The value is derived from the real-world value **Min**. This property is available only in the property dialog box.

**Stored Int Max**

For Simulink.Breakpoint objects with a fixed-point data type, the maximum value of the elements in the breakpoint set, specified as a stored integer value. The value is derived from the real-world value **Max**. This property is available only in the property dialog box.

**Unit**

Physical unit of the elements in the breakpoint set. You can specify text such as degC. See “Unit Specification in Simulink Models”.

**Field name**

Name of a structure field in the generated code. This field stores the breakpoint set data. The default value is **BP**. To change the field name, specify text.

This column appears only if you select **Support tunable size**.

**Tunable size name**

Name of a structure field in the generated code. This scalar field stores the length of the breakpoint set (the number of elements), which the generated code algorithm uses to determine the size of the table. To tune the effective size of the table during code execution, change the value of this structure field in memory. The default name is **N**. To change the field name, specify text.

This column appears only if you select **Support tunable size**.

**Description**

Description of the breakpoint set. You can specify text such as **This breakpoint set represents the pressure input**.

**Data definition: Storage class**

Storage class of the structure variable (if you select **Support tunable size**) or array variable in the generated code. The variable stores the breakpoint set data. The default setting is **Auto**.

For more information about storage classes, see “Apply Storage Classes to Individual Signal, State, and Parameter Data Elements” (Simulink Coder).
If you have Embedded Coder, you can choose a custom storage class. For information about custom storage classes, see “Apply Built-In and Customized Storage Classes to Data Elements” (Embedded Coder).

**Data definition: Alias**

Alternative name for the variable in the generated code. The default value is empty, in which case the generated code uses the name of the Simulink.Breakpoint object as the name of the variable. To set the alias, specify text.

To enable this property, set **Data definition: Storage class** to a setting other than Auto.

**Data definition: Alignment**

Data alignment boundary in the generated code. The starting memory address for the data allocated for the structure or array variable is a multiple of the value that you specify. The default value is \(-1\), which allows the code generator to determine an optimal alignment based on usage.

Specify a positive integer that is a power of 2, not exceeding 128. For more information about using data alignment for code replacement, see “Data Alignment for Code Replacement” (Embedded Coder).

**Struct Type definition: Name**

Name of the structure type that the structure variable uses in the generated code. The default value is empty. Specify text.

This property appears only if you select **Support tunable size**.

**Struct Type definition: Data scope**

Scope of the structure type definition (imported from your handwritten code or exported from the generated code). The default value is Auto. When you select Auto:

- If you do not specify a value in the **Struct Type definition: Header file** box, the generated code exports the structure type definition to the file `model_types.h`. `model` is the name of the model.
- If you specify a value in the **Struct Type definition: Header file** box, such as `myHdr.h`, the generated code imports the structure type definition from `myHdr.h`.

To explicitly specify the data scope:

- To import the structure type definition into the generated code from your custom code, select **Imported**.
• To export the structure type definition from the generated code, select Exported.

If you do not specify a value in the **Struct Type definition: Header file** box, the generated code imports or exports the type definition from or to **StructName.h**. **StructName** is the name that you specify with the property **Struct Type definition: Name**.

This property appears only if you select **Support tunable size**.

**Struct Type definition: Header file**

Name of the header file that contains the structure type definition. You can import the definition from a header file that you create, or export the definition into a generated header file. To control the scope of the structure type, adjust the setting for the **Struct Type definition: Data scope** property.

This property appears only if you select **Support tunable size**.

**Properties**

**Breakpoints — Breakpoint set data**

*Simulink.lookuptable.Breakpoint* object

Breakpoint set data, specified as a *Simulink.lookuptable.Breakpoint* object. Use this embedded object to configure the structure field names and characteristics of the breakpoint set data such as breakpoint values, data type, and dimensions.

**CoderInfo — Code generation settings for variable**

*Simulink.CoderInfo* object

Code generation settings for the structure variable (if you set **SupportTunableSize** to true) or array variable (false) that stores the breakpoint set data, specified as a *Simulink.CoderInfo* object. You can specify a storage class or custom storage class by using this embedded object. For more information, see *Simulink.CoderInfo*.

**StructTypeInfo — Settings for structure type in the generated code**

*Simulink.lookuptable.StructTypeInfo* object

Settings for the structure type that the structure variable uses in the generated code, specified as a *Simulink.lookuptable.StructTypeInfo* object.
If you set `SupportTunableSize` to `false`, the `Simulink.Breakpoint` object does not appear in the generated code as a structure. The code generator ignores this property.

**SupportTunableSize** — Option to generate code that enables tunability of table size  

false (default) | true

Option to generate code that enables tunability of the effective size of the table, specified as `true` or `false`. See the **Support Tunable Size** parameter.

Data Types: `logical`

## Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects (MATLAB).

## Examples

### Share Breakpoint Data Between One-Dimensional Lookup Tables

   ```matlab
   myBpSet = Simulink.Breakpoint
   ```
2. Specify the breakpoint data.
   ```matlab
   myBpSet.Breakpoints.Value = [-2 -1 0 1 2];
   ```
3. Create a `Simulink.LookupTable` object named `FirstLUTObj`.
   ```matlab
   FirstLUTObj = Simulink.LookupTable;
   ```
4. Specify the table data.
   ```matlab
   FirstLUTObj.Table.Value = [1.1 2.2 3.3 4.4 5.5];
   ```
5. Configure the lookup table object to refer to the breakpoint set object.
   ```matlab
   FirstLUTObj.Breakpoints = {'myBpSet'};
   ```
Create another `Simulink.LookupTable` object to store a different set of table data. Configure the lookup table object to refer to the same breakpoint set object.

```matlab
SecondLUTObj = Simulink.LookupTable;
SecondLUTObj.Table.Value = [1.2 2.3 3.4 4.5 5.6];
SecondLUTObj.Breakpoints = {'myBpSet'};
```

You can use `FirstLUTObj` and `SecondLUTObj` to specify the table data in two different Interpolation Using Prelookup blocks. Use `myBpSet` to specify the breakpoint set data in one or two Prelookup blocks that provide the inputs for the Interpolation Using Prelookup blocks.

**Limitations**

- You cannot subclass `Simulink.Breakpoint` or `Simulink.LookupTable`. For this reason, you cannot apply custom storage classes other than those in the built-in Simulink package.
- You cannot use `Simulink.Breakpoint` objects or `Simulink.LookupTable` objects that refer to `Simulink.Breakpoint` objects as instance-specific parameter data for reusable components. For example, you cannot use one of these objects as:
  
  - A model argument in a model workspace or a model argument value in a Model block.
  - The value of a mask parameter on a CodeReuse Subsystem block.
  - The value of a mask parameter on a subsystem that you reuse by creating a custom library.

  However, you can use standalone `Simulink.LookupTable` objects, which do not refer to `Simulink.Breakpoint` objects, in these ways.

- You cannot generate code according to the FIX AXIS style.
- When blocks in a subsystem use `Simulink.LookupTable` or `Simulink.Breakpoint` objects, you cannot set data type override (see “Control Fixed-Point Instrumentation and Data Type Override”) only on the subsystem. Instead, set data type override on the entire model.
See Also
Simulink.LookupTable | Simulink.Parameter |
Simulink.lookuptable.Breakpoint | Simulink.lookuptable.StructTypeInfo |
Simulink.lookuptable.Table

Topics
“Configure Lookup Tables for AUTOSAR Measurement and Calibration” (AUTOSAR Blockset)
“About Lookup Table Blocks”
“Package Shared Breakpoint and Table Data for Lookup Tables”
“Create Tunable Calibration Parameter in the Generated Code” (Simulink Coder)

Introduced in R2016b
Simulink.CoderInfo

Specify information needed to generate code for signal, state, or parameter data

Description

Use a Simulink.CoderInfo object to specify code generation settings for signal, state, and parameter data in a model.

Simulink creates a Simulink.CoderInfo object for each data object that you create. Data objects represent signal, state, or parameter data. The Simulink.CoderInfo object exists in the CoderInfo property of each data object.

Data objects include objects of these classes:

- Simulink.Parameter
- Simulink.Signal
- Simulink.LookupTable
- Simulink.Breakpoint
- Simulink.DualScaledParameter

Use the properties of the Simulink.CoderInfo object to configure the representation of the parent data object in the generated code.

You can set the properties of a Simulink.CoderInfo object through the CoderInfo property or the property dialog box of the parent data object. For example, the following MATLAB expression sets the StorageClass property of a Simulink.CoderInfo object used by a signal object named mysignal.

```
mysignal.CoderInfo.StorageClass = 'ExportedGlobal';
```

Creation

When you create a data object, Simulink sets the value of the CoderInfo property by creating a Simulink.CoderInfo object. You do not need to create a Simulink.CoderInfo object explicitly.
Properties

**Alias — Alternative name for code generation**

' ' (empty character vector) (default) | character vector

Alternative name for the data in the generated code, specified as a character vector.

Example: 'myOtherName'

Data Types: char

**Alignment — Data alignment boundary**

-1 (default) | positive integer

Data alignment boundary for this data, specified as a positive integer that is a power of 2, not exceeding 128. Specify an integer number of double data type. See “Data Alignment for Code Replacement” (Embedded Coder) for more information.

Example: 8

Data Types: double
Complex Number Support: Yes

**CustomAttributes — Custom storage class attributes of this data**

SimulinkCSC.AttribClass_Simulink_Default object (default) | custom attributes object

Custom storage class attributes of this data, returned as a custom attributes object. You must set the property StorageClass to 'Custom' to enable this property.

Depending on the custom storage class that you apply by using the CustomStorageClass property of the Simulink.CoderInfo object, Simulink sets the value of this property by creating a custom attributes object. Then, you can set the values of the properties of the custom attributes object. See “Apply Built-In and Customized Storage Classes to Data Elements” (Embedded Coder) for more information.

**CustomStorageClass — Custom storage class of this data**

'Default' (default) | character vector

Custom storage class of this data, specified as a character vector. You must set the property StorageClass to 'Custom' to enable this property.
For a list of valid custom storage classes (Embedded Coder) when you create the data object from the Simulink package, see “Choose Storage Class for Controlling Data Representation in Generated Code” (Embedded Coder).

Example: 'ExportToFile'

Data Types: char

**StorageClass — Storage class of this data**

'Auto' (default) | character vector

Storage class of this data, specified as a character vector. For more information, see “Apply Storage Classes to Individual Signal, State, and Parameter Data Elements” (Simulink Coder).

Example: 'ExportedGlobal'

Data Types: char

### Examples

**Configure Code Generation Settings Programmatically**

For examples that show how to configure code generation settings for a data item programmatically, see “Apply Storage Classes to Individual Signal, State, and Parameter Data Elements” (Simulink Coder) and “Apply Built-In and Customized Storage Classes to Data Elements” (Embedded Coder).

### See Also

**Topics**

“Apply Storage Classes to Individual Signal, State, and Parameter Data Elements” (Simulink Coder)

“Create Tunable Calibration Parameter in the Generated Code” (Simulink Coder)

“Data Objects”

“Apply Built-In and Customized Storage Classes to Data Elements” (Embedded Coder)

**Introduced in R2015a**
Simulink.ConfigSetRef

Link model to configuration set stored independently of any model

Description

Instances of this handle class allow a model to reference configuration sets that exist outside any model. See “Manage a Configuration Set”, “Overview”, and “Manage a Configuration Reference” for more information.

Property Summary

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Description” on page 5-257</td>
<td>Description of the configuration reference.</td>
</tr>
<tr>
<td>“Name” on page 5-257</td>
<td>Name of the configuration reference.</td>
</tr>
<tr>
<td>“SourceName” on page 5-258</td>
<td>Name of the variable in the workspace or the data dictionary that contains the referenced configuration set.</td>
</tr>
</tbody>
</table>

Note You can use the Configuration Reference dialog box to set the Name, Description, and SourceName properties of a configuration reference. See “Create and Attach a Configuration Reference” for details.

Method Summary

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>“copy” on page 5-258</td>
<td>Create a copy of a configuration reference.</td>
</tr>
<tr>
<td>“getFullName” on page 5-259</td>
<td>Get the full pathname of a configuration reference.</td>
</tr>
<tr>
<td>“getModel” on page 5-259</td>
<td>Get the handle of the model that owns a configuration reference.</td>
</tr>
</tbody>
</table>
### Properties

#### Description

**Description**

Description of the configuration reference. You can use this property to provide additional information about a configuration reference, such as its purpose. This field can remain blank.

**Data Type**

character vector

**Access**

RW

#### Name

**Description**

Name of the configuration reference. This name represents the configuration reference in the GUI.

**Data Type**

character vector
Access
RW

SourceName

Description
Name of the variable in the workspace or the data dictionary that contains the referenced configuration set.

Data Type
character vector

Access
RW

Methods

copy

Purpose
Create a copy of this configuration reference.

Syntax
copy

description
This method creates a copy of this configuration set.

Note You must use this method to create copies of configuration references. This is because Simulink.ConfigSetRef is a handle class. See “Handle Versus Value Classes” for more information.
**getFullName**

**Purpose**
Get the full pathname of a configuration reference.

**Syntax**
getFullName

**Description**
This method returns a character vector specifying the full pathname of a configuration reference, e.g., 'vdp/Configuration'.

**getModel**

**Purpose**
Get the model that owns this configuration reference.

**Syntax**
getModel

**Description**
Returns a handle to the model that owns this configuration reference.

**example**
The following command opens the block diagram of the model that owns the configuration set referenced by the MATLAB workspace variable hCr.

```matlab
open_system(hCr.getModel);
```

**get_param**

**Purpose**
Get the value of a configuration set parameter indirectly through a configuration reference.
Syntax

get_param(paramName)

Arguments

paramName

Character vector specifying the name of the parameter whose value is to be returned.

Description

This method returns the value of the specified parameter from the configuration set to which the configuration reference points. To obtain this value, the method uses the value of SourceName to retrieve the configuration set, then retrieves the value of paramName from that configuration set. Specifying paramName as 'ObjectParameters' returns the names of all valid parameters in the configuration set. If a valid configuration set is not attached to the configuration reference, the method returns unreliable values.

The inverse method, set_param, is not defined for configuration references. To obtain a parameter value through a configuration reference, you must first use the getRefConfigSet method to retrieve the configuration set from the reference, then use set_param directly on the configuration set itself.

You can also use the get_param model construction command to get the values of parameters of a model's active configuration set, e.g., get_param(bdroot, 'SolverName') gets the solver name of the currently selected model.

eexample

The following command gets the name of the solver used by the selected model's active configuration.

    hAcs = getActiveConfigSet(bdroot);
    hAcs.get_param('SolverName');

getRefConfigSet

Purpose

Get the configuration set specified by a configuration reference
**Syntax**

getRefConfigSet

**Description**

Returns a handle to the configuration set specified by the `SourceName` property of a configuration reference.

**isActive**

**Purpose**

Determine whether this configuration set is its model's active configuration set.

**Syntax**

isActive

**Description**

Returns `true` if this configuration set is the active configuration set of the model that owns this configuration set.

**refresh**

**Purpose**

Update configuration reference after any change to properties or configuration set availability

**Syntax**

refresh

**Description**

Updates a configuration reference after using the API to change any property of the reference, or after providing a configuration set that did not exist at the time the set was originally specified in `SourceName`. If you omit executing `refresh` after any such change, the configuration reference handle will be stale, and using it will give incorrect results.
Introduced in R2007a
Simulink.FindOptions class

Package: Simulink

Options for finding blocks in models and subsystems

Description

Create an options object to use with Simulink.findBlocks and Simulink.findBlocksOfType to constrain the search.

Construction

f = Simulink.FindOptions creates a FindOptions object that uses the default search options.

f = Simulink.FindOptions(Option1,Value1,...OptionN,ValueN) creates the object using the specified search options.

Input Arguments

'CaseSensitive' — Option to specify whether to match case when searching
true (default) | false

Option to specify whether to match case when searching, specified as true for case-sensitive search or false for case-insensitive search.

'FollowLinks' — Option to follow library links
false (default) | true

Option to follow library links, specified as true or false. If true, search follows links into library blocks.

'IncludeCommented' — Option for the search to include commented blocks
true (default) | false

Option for the search to include commented blocks, specified as true or false.
'LookUnderMasks' — Options to search masked blocks
'all' (default) | 'none' | 'functional' | 'graphical'

Options to search masked blocks, specified as:

- 'all' — Search in all masked blocks.
- 'none' — Prevent searching in masked systems.
- 'functional' — Include masked subsystems that do not have dialogs.
- 'graphical' — Include masked subsystems that do not have workspaces or dialogs.

'Variants' — Options to search Variant subsystems
'AllVariants' (default) | 'ActiveVariants' | 'ActivePlusCodeVariants'

Options to search Variant subsystems, specified as:

- 'AllVariants' — Search all variant choices.
- 'ActiveVariants' — Search only active variant choices.
- 'ActivePlusCodeVariants' — Search all variant choices with 'Generate preprocessor conditionals' active. Otherwise, search only the active variant choices.

The 'Variants' search constraint applies only to variant subsystems and model variants.

'RegExp' — Option to treat the search text as a regular expression
false (default) | true

Option to treat the search text as a regular expression, specified as true or false. To learn more about MATLAB regular expressions, see “Regular Expressions” (MATLAB).

'SearchDepth' — Levels in the model to search
positive integer

Levels in the model to search, specified as a positive integer. The default is to search all levels. Specify:

- 1 — Search in the top-level system.
- 2 — Search the top-level system and its children, 3 to search an additional level, and so on.
Examples

Specify Search Options to Use with Simulink.findBlocks

Search for all blocks in the Unlocked subsystem but not in any of its children.

```matlab
f = Simulink.FindOptions('SearchDepth',1);
load_system('sldemo_clutch');
bl = Simulink.findBlocks('sldemo_clutch/Unlocked',f)
```

```matlab
tbl = Simulink.findBlocks('sldemo_clutch/Unlocked',f)
```

To get the block name and path instead of the handle, use `getfullname`.

```matlab
bl = getfullname (Simulink.findBlocks('sldemo_clutch/Unlocked',f))
```

```matlab
bl =
20×1 cell array
    {'sldemo_clutch/Unlocked/Tfmaxk'
```
See Also

Simulink.allBlockDiagrams | Simulink.findBlocks |
Simulink.findBlocksOfType

Introduced in R2018a
Simulink.GlobalDataTransfer class

Package: Simulink

Configure concurrent execution data transfers

Description

The Simulink.GlobalDataTransfer object contains the data transfer information for the concurrent execution of a model. To access the properties of this class, use the get_param function to get the handle for this class, and then use dot notation to access the properties. For example:

```matlab
dt=get_param(gcs,'DataTransfer');
dt.DefaultTransitionBetweenContTasks
ans =
Ensure deterministic transfer (minimum delay)
```

Properties

DefaultTransitionBetweenSyncTasks

Global setting for data transfer handling option when the source and destination of a signal are in two different and periodic tasks.

Data Type: Enumeration. Can be one of:

- 'Ensure data integrity only'
- 'Ensure deterministic transfer (maximum delay)'
- 'Ensure deterministic transfer (minimum delay)'

Access: Read/write
**DefaultTransitionBetweenContTasks**

Global setting for the data transfer handling option for signals that have a continuous sample time.

Data Type: Enumeration. Can be one of:
- 'Ensure data integrity only'
- 'Ensure deterministic transfer (maximum delay)'
- 'Ensure deterministic transfer (minimum delay)'

Access: Read/write

**DefaultExtrapolationMethodBetweenContTasks**

Global setting for the data transfer extrapolation method for signals that have a continuous sample time.

Data Type: Enumeration. Can be one of:
- 'None'
- 'Zero Order Hold'
- 'Linear'
- 'Quadratic'

Access: Read/write

**Copy Semantics**

Handle. To learn how handle classes affect copy operations, see Copying Objects (MATLAB).

**Examples**

Access the properties of this class.

```matlab
dt=get_param(gcs,'DataTransfer');
dt.DefaultTransitionBetweenContTasks```
ans =

Ensure deterministic transfer (minimum delay)

**See Also**

Simulink.architecture.add | Simulink.architecture.delete |
Simulink.architecture.find_system | Simulink.architecture.get_param |
Simulink.architecture.importAndSelect | Simulink.architecture.profile |
Simulink.architecture.register | Simulink.architecture.set_param

**Topics**

“Configure Data Transfer Settings Between Concurrent Tasks”
Simulink.LookUpTable class

**Package:** Simulink

Store and share lookup table and breakpoint data, configure the data for ASAP2 and AUTOSAR code generation

**Description**

An object of the Simulink.LookUpTable class stores lookup table and breakpoint data. You can use that data in a lookup table block such as the n-D Lookup Table block. With the object, you can specify data types and code generation settings for the table and the breakpoint sets.

When you store all of the table and breakpoint set data in a single Simulink.LookUpTable object, all of the data appears in a single structure in the generated code. To configure STD_AXIS code generation for calibration, use this technique.

To share a breakpoint set between multiple lookup tables, for example for COM_AXIS code generation, use a Simulink.Breakpoint object in one or more Prelookup blocks. Use Simulink.LookUpTable objects in Interpolation Using Prelookup blocks. Then, configure the lookup table objects to refer to the breakpoint object. For more information, see “Package Shared Breakpoint and Table Data for Lookup Tables”.

**Construction**

LUTObj = Simulink.LookUpTable returns a Simulink.LookUpTable object LUTObj with default property values.

To create a Simulink.LookUpTable object by using the Model Explorer, use the button on the toolbar. The default name for the object is Object.
Property Dialog Box

**Number of table dimensions**
Number of dimensions of the lookup table. Specify an integer value up to 30 (inclusive). For example, to represent a three-dimensional lookup table, specify the integer 3.

**Table**
Information for the table data. You can configure these characteristics:

**Value**
Table data. Specify a numeric vector or multidimensional array with at least two elements.
You can also use an expression with mathematical operators such as 
\[ \sin(1:0.5:30) \] as long as the expression returns a numeric vector or
multidimensional array. When you click **Apply** or **OK**, the object executes the
expression and uses the result to set the value of this property.

When you set **Data type** to **auto**, to set **Value**, use a typed expression such as
\[ \text{single}([1 2 3]) \] or use the \texttt{fi} constructor to embed an \texttt{fi} object.

When you specify table data with three or more dimensions, **Value** displays the
data as an expression that contains a call to the \texttt{reshape} function. To edit the
values in the data, modify the first argument of the \texttt{reshape} call, which contains
all of the values in a serialized vector. When you add or remove elements along a
dimension, you must also correct the argument that represents the length of the
modified dimension.

You can edit this data by using a more intuitive interface in a lookup table block.
See “Import Lookup Table Data from MATLAB”.

**Data type**

Data type of the table data. The default setting is **auto**, which means that the
table data acquire a data type from the value that you specify in **Value**. If you use
an untyped expression such as \([1 2 3]\) to set **Value**, the table data use the data
type **double**. If you specify a typed expression such as \(\text{single}([1 2 3])\) or an
\texttt{fi} object, the table data use the data type specified by the expression or object.
Enumerated data types are also supported.

You can explicitly specify an integer data type, a floating-point data type, a fixed-
point data type, or a data type expression such as the name of a
\texttt{Simulink.AliasType} object.

For more information about data types in Simulink, see “Data Types Supported by
Simulink”. To decide how to control the data types of table and breakpoint data in
\texttt{Simulink.LookupTable} and \texttt{Simulink.Breakpoint} objects, see “Control
Data Types of Lookup Table Objects” (Simulink Coder).

**Dimensions**

Dimension lengths of the lookup table data.

To use symbolic dimensions, specify a character vector. See “Implement
Dimension Variants for Array Sizes in Generated Code” (Embedded Coder).
Min
Minimum value of the elements in the table data. The default value is empty, [ ].
You can specify a numeric, real value.

For more information about how Simulink uses this property, see “Specify
Minimum and Maximum Values for Block Parameters”.

Max
Maximum value of the elements in the table data. The default value is empty, [ ].
You can specify a numeric, real value.

For more information about how Simulink uses this property, see “Specify
Minimum and Maximum Values for Block Parameters”.

Stored Int Min
For Simulink.LookupTable objects with a fixed-point data type, the minimum
value of the elements in the table data, specified as a stored integer value. The
value is derived from the real-world value Min. This property is available only in
the property dialog box.

Stored Int Max
For Simulink.LookupTable objects with a fixed-point data type, the maximum
value of the elements in the table data, specified as a stored integer value. The
value is derived from the real-world value Max. This property is available only in
the property dialog box.

Unit
Physical units of the elements in the lookup table. You can specify text such as
degC. See “Unit Specification in Simulink Models”.

Field name
Name of a structure field in the generated code. This field stores the table data if
you configure the Simulink.LookupTable object to appear in the generated
code as a structure. The default value is Table. To change the field name, specify
text.

Description
Description of the lookup table. You can specify text such as This lookup
table describes the action of a pump.

Breakpoints
Breakpoint set information. Each row is one breakpoint set. To configure additional
breakpoint sets, specify an integer value in the Number of table dimensions box.
For the breakpoint sets, you can configure these characteristics:

**Specification**

Source for the information of the breakpoint sets, specified as *Explicit values* (default), *Reference*, or *Even spacing*.

- To store all of the table and breakpoint set data in the `Simulink.LookupTable` object, set **Specification** to *Explicit values*.

  The `Simulink.LookupTable` object appears in the generated code as a single structure variable.

- To store the table data in the `Simulink.LookupTable` object and store the breakpoint set data in `Simulink.Breakpoint` objects, set **Specification** to *Reference*.

  The `Simulink.LookupTable` object appears in the generated code as a separate array variable that contains the table data. Each `Simulink.Breakpoint` object appears as a separate array or structure variable that contains the breakpoint set data.

- To store the table data and evenly spaced breakpoints in the `Simulink.LookupTable` object, set **Specification** to *Even spacing*. Use the **First point** and **Spacing** parameters to generate a set of evenly spaced breakpoints.

**Note** When **Specification** is set to *Explicit values* or *Even spacing*, you can change the order of the tunable size, breakpoint, and table entries in a lookup table object-generated structure.

**Support tunable size**

Specification to enable tuning the effective size of the table in the generated code. In the code, the structure that corresponds to the object has an extra field for each breakpoint vector. Each extra field stores the length of the corresponding breakpoint vector. You can change the value of each field to adjust the effective size of the table.

This property appears only if you set **Specification** to *Explicit values* or *Even spacing*. 

5-274
**Note** If you store breakpoint data in `Simulink.Breakpoint` objects by setting `Specification` to `Reference`, to enable tuning of the table size in the generated code, use the **Support tunable size** property of each `Simulink.Breakpoint` object.

**Value**

Data for the breakpoint set. Specify a numeric vector with at least two elements.

You can also use an expression with mathematical operators such as \( \sin(1:0.5:30) \) as long as the expression returns a numeric vector. When you click **Apply** or **OK**, the object executes the expression and uses the result to set the value of this property.

When you set **Data type** to `auto`, to set **Value**, use a typed expression such as `single([1 2 3])` or use the `fi` constructor to embed an `fi` object.

You can edit this data by using a more intuitive interface in a lookup table block. See “Import Lookup Table Data from MATLAB”.

**Data type**

Data type of the breakpoint set. The default setting is `auto`, which means that the breakpoint set acquires a data type from the value that you specify in **Value**. If you use an untyped expression such as `[1 2 3]` to set **Value**, the breakpoint data use the data type `double`. If you specify a typed expression such as `single([1 2 3])` or an `fi` object, the breakpoint data use the data type specified by the expression or object.

You can explicitly specify an integer data type, a floating-point data type, a fixed-point data type, or a data type expression such as the name of a `Simulink.AliasType` object.

For more information about data types in Simulink, see “Data Types Supported by Simulink”. To decide how to control the data types of table and breakpoint data in `Simulink.LookupTable` and `Simulink.Breakpoint` objects, see “Control Data Types of Lookup Table Objects” (Simulink Coder).

**Dimensions**

Dimension lengths of the breakpoint set.

To use symbolic dimensions, specify a character vector. See “Implement Dimension Variants for Array Sizes in Generated Code” (Embedded Coder).
Min
Minimum value of the elements in the breakpoint set. The default value is empty, [ ]. You can specify a numeric, real value.

For more information about how Simulink uses this property, see “Specify Minimum and Maximum Values for Block Parameters”.

Max
Maximum value of the elements in the breakpoint set. The default value is empty, [ ]. You can specify a numeric, real value.

For more information about how Simulink uses this property, see “Specify Minimum and Maximum Values for Block Parameters”.

Unit
Physical unit of the elements in the breakpoint set. You can specify text such as degF. See “Unit Specification in Simulink Models”.

Field name
Name of a structure field in the generated code. This field stores the breakpoint set data. The default value is BP1 for the first breakpoint set and BP2 for the second set. To change the field name, specify text.

Tunable size name
Name of a structure field in the generated code. This field stores the length (number of elements) of the breakpoint set, which the generated code algorithm uses to determine the size of the table. To tune the effective size of the table during code execution, change the value of this structure field in memory. The default name is N1 for the first breakpoint set and N2 for the second set. To change the field name, specify text.

This column appears only if you select Support tunable size.

Description
Description of the breakpoint set. You can specify text such as This breakpoint set represents the pressure input.

First point
First point in evenly spaced breakpoint data. This parameter is available when Specification is set to Even spacing.
Spacing

Spacing between points in evenly spaced breakpoint data. This parameter is available when **Specification** is set to **Even spacing**.

**Name**

Name of the Simulink.Breakpoint object that stores the information for this breakpoint set.

This column appears only if you set **Specification** to **Reference**.

**First point name**

Name of the Simulink.Breakpoint object that stores the information for the first point. This parameter is available when **Specification** is set to **Even spacing**.

**Spacing name**

Name of the Simulink.Breakpoint object that stores the information for the spacing. This parameter is available when **Specification** is set to **Even spacing**.

**Data definition: Storage class**

Storage class of the structure variable (if you set **Specification** to **Explicit values** or **Even spacing**) or array variable (Reference) in the generated code. The variable stores the table data and, if the variable is a structure, the breakpoint set data. The default setting is **Auto**.

For more information about storage classes, see “Apply Storage Classes to Individual Signal, State, and Parameter Data Elements” (Simulink Coder).

If you have Embedded Coder, you can choose a custom storage class. For information about custom storage classes, see “Apply Built-In and Customized Storage Classes to Data Elements” (Embedded Coder).

**Data definition: Alias**

Alternative name for the structure variable (if you set **Specification** to **Explicit values** or **Even spacing**) or array variable (Reference) in the generated code. The default value is empty, in which case the generated code uses the name of the Simulink.LookupTable object as the name of the structure or array variable. To set the alias, specify text.

To enable this property, set **Data definition: Storage class** to a setting other than **Auto**.
Data definition: Alignment

Data alignment boundary in the generated code. The starting memory address for the
data allocated for the structure or array variable is a multiple of the value that you
specify. The default value is -1, which allows the code generator to determine an
optimal alignment based on usage.

Specify a positive integer that is a power of 2, not exceeding 128. For more
information about using data alignment for code replacement, see “Data Alignment
for Code Replacement” (Embedded Coder).

Struct Type definition: Name

Name of the structure type that the structure variable uses in the generated code.
The default value is empty. Specify text.

This property appears only if you set Specification to Explicit values or Even
spacing.

Struct Type definition: Data scope

Scope of the structure type definition (imported from your custom code or exported
from the generated code). The default value is Auto. When you select Auto:

- If you do not specify a value in the Struct Type definition: Header file box, the
generated code exports the structure type definition to the file model_types.h.
  model is the name of the model.
- If you specify a value in the Struct Type definition: Header file box, such as
  myHdr.h, the generated code imports the structure type definition from myHdr.h.

To explicitly specify the data scope:

- To import the structure type definition into the generated code from your custom
code, select Imported.
- To export the structure type definition from the generated code, select Exported.

If you do not specify a value in the Struct Type definition: Header file box, the
generated code imports or exports the type definition from or to StructName.h.
StructName is the name that you specify by using the property Struct Type
definition: Name.

This property appears only if you set Specification to Explicit values or Even
spacing.
Struct Type definition: Header file

Name of the header file that contains the structure type definition. You can import the definition from a header file that you create, or export the definition into a generated header file. To control the scope of the structure type, adjust the setting for the Struct Type definition: Data scope property.

This property appears only if you set Specification to Explicit values or Even spacing.

Properties

Breakpoints — Breakpoint set information
vector of Simulink.lookuptable.Breakpoint objects | cell array of character vectors

Breakpoint set information, specified as a vector of Simulink.lookuptable.Breakpoint objects, a cell array of character vectors, or a vector of Simulink.lookuptable.Evenspacing objects.

If you use a vector of Simulink.lookuptable.Breakpoint objects, each object represents a breakpoint set. To use a vector of Simulink.lookuptable.Breakpoint objects, set the property BreakpointsSpecification to 'Explicit values'.

If you use a cell array of character vectors, each character vector represents the name of a Simulink.Breakpoint object. To use a cell array of character vectors, set the property BreakpointsSpecification to 'Reference'.

If you use a vector of Simulink.lookuptable.Evenspacing objects, each object represents a breakpoint set. To use a vector of Simulink.lookuptable.Evenspacing objects, set the property BreakpointsSpecification to 'Even Spacing'.

BreakpointsSpecification — Source of breakpoint set information
'Explicit values' (default) | 'Reference' | 'Even spacing'

Source of the breakpoint set information, specified as 'Explicit values' (default), 'Even spacing', or 'Reference'. See the Breakpoints > Specification parameter.

Data Types: char

CoderInfo — Code generation settings for structure or array variable
Simulink.CoderInfo object
Code generation settings for the structure variable (if you set BreakpointsSpecification to 'Explicit values' or 'Even spacing') or array variable ('Reference') that stores the lookup table and breakpoint sets, specified as a Simulink.CoderInfo object. You can specify a storage class or custom storage class by using this embedded object. See Simulink.CoderInfo.

**StructTypeInfo — Settings for structure type in the generated code**

Simulink.lookuptable.StructTypeInfo object

Settings for the structure type that the structure variable uses in the generated code, specified as a Simulink.lookuptable.StructTypeInfo object.

If you set BreakpointsSpecification to 'Reference', the Simulink.LookupTable object does not appear in the generated code as a structure. The code generator ignores this property.

**SupportTunableSize — Option to generate code that enables tunability of table size**

false (default) | true

Option to generate code that enables tunability of the effective size of the table, specified as true or false. See the **Support Tunable Size** parameter.

Data Types: logical

**Table — Information for table data**

Simulink.lookuptable.Table object

Information for the table data, specified as a Simulink.lookuptable.Table object.

**Copy Semantics**

Handle. To learn how handle classes affect copy operations, see Copying Objects (MATLAB).
Examples

Represent a One-Dimensional Lookup Table

1. Create a Simulink.LookupTable object named LUTObj.
   
   LUTObj = Simulink.LookupTable;

2. Specify the table data.
   
   LUTObj.Table.Value = [1.1 2.2 3.3 4.4 5.5];

3. Specify the breakpoint set data.
   
   LUTObj.Breakpoints(1).Value = [-2 -1 0 1 2];

4. Specify a name for the structure type in the generated code.
   
   LUTObj.StructTypeInfo.Name = 'myLUTStruct';

You can use LUTObj in a 1-D Lookup Table block dialog box. In the block, set Data specification to Lookup table object and Name to LUTObj.

Represent a Two-Dimensional Lookup Table

1. Create a Simulink.LookupTable object named LUTObj.
   
   LUTObj = Simulink.LookupTable;

2. Specify the table data.
   
   LUTObj.Table.Value = [1.1 2.2 3.3 4.4 5.5; ... 
                        6.6 7.7 8.8 9.9 10.1];

3. Specify the breakpoint set data. In the Breakpoints property, use the vector index 2 to set the values in the second breakpoint set.
   
   LUTObj.Breakpoints(1).Value = [-1 1];

   LUTObj.Breakpoints(2).Value = [-2 -1 0 1 2];

   LUTObj creates a Simulink.lookuptable.Breakpoint object as the second vector element in the value of the Breakpoints property. Except for the Value property, the new object has default property values.
Specify a name for the structure type in the generated code.

LUTObj.StructTypeInfo.Name = 'myLUTStruct';

You can use LUTObj in a 2-D Lookup Table block dialog box.

**Evenly Space Every Second Value Starting from 1**

To evenly space every second value starting from 1, use the Breakpoint object.

1. Create a Simulink.LookupTable object named LUTObj.

   ```
   LUTObj=Simulink.LookupTable
   ```

   LUTObj =

   LookupTable with properties:

   Table: [1x1 Simulink.lookuptable.Table]
   BreakpointsSpecification: 'Explicit values'
   Breakpoints: [1x1 Simulink.lookuptable.Breakpoint]
   SupportTunableSize: 0
   CoderInfo: [1x1 Simulink.CoderInfo]
   StructTypeInfo: [1x1 Simulink.lookuptable.StructTypeInfo]

2. Set up the breakpoint property to even spacing.

   ```
   LUTObj.BreakpointsSpecification='Even spacing'
   ```

   LUTObj =

   LookupTable with properties:

   Table: [1x1 Simulink.lookuptable.Table]
   BreakpointsSpecification: 'Even spacing'
   Breakpoints: [1x1 Simulink.lookuptable.Evenspacing]
   SupportTunableSize: 0
   CoderInfo: [1x1 Simulink.CoderInfo]
   StructTypeInfo: [1x1 Simulink.lookuptable.StructTypeInfo]

3. Get the properties of the breakpoint.

   ```
   LUTObj.Breakpoints(1)
   ```
ans =

Evenspacing with properties:

    FirstPoint: 0
    Spacing: 1
    DataType: 'auto'
        Min: []
        Max: []
        Unit: ''
    FirstPointName: 'BPFirstPoint1'
    SpacingName: 'BPSpacing1'
    TunableSizeName: 'N1'
        Description: ''

4 To set the first point property, use the Breakpoint object FirstPoint property.
LUTObj.Breakpoints(1).FirstPoint=1

5 To set the spacing property, use the Breakpoint object.
LUTObj.Breakpoints(1).Spacing=2

6 Get the properties of the breakpoint.
LUTObj.Breakpoints(1)

ans =

Evenspacing with properties:

    FirstPoint: 1
    Spacing: 2
    DataType: 'auto'
        Min: []
        Max: []
        Unit: ''
    FirstPointName: 'BPFirstPoint1'
    SpacingName: 'BPSpacing1'
    TunableSizeName: 'N1'
        Description: ''
Control Code Generation for Lookup Table and Breakpoint Sets

Create a Simulink.LookupTable object named LUTObj.

LUTObj = Simulink.LookupTable;

Specify the table data.

LUTObj.Table.Value = [1.00 2.25 3.50 4.75 6.00; ... 7.25 8.50 9.75 11.00 12.25];

Specify the breakpoint set data. In the Breakpoints property, use the array index 2 to create an additional Simulink.LookupTable.BreakpointInfo object, which represents the second breakpoint set.

LUTObj.Breakpoints(1).Value = [-1 1];
LUTObj.Breakpoints(2).Value = [-2 -1 0 1 2];

Specify data types for the lookup table and each breakpoint set.

LUTObj.Table.DataType = 'fixdt(1,16,2)';
LUTObj.Breakpoints(1).DataType = 'int16';
LUTObj.Breakpoints(2).DataType = 'int16';

Specify unique names for the structure fields that store the table data and breakpoint sets in the generated code.

LUTObj.Table.FieldName = 'myTable';
LUTObj.Breakpoints(1).FieldName = 'myBPSet1';
LUTObj.Breakpoints(2).FieldName = 'myBPSet2';

Export the structure variable definition from the generated code by using the storage class ExportedGlobal.

LUTObj.CoderInfo.StorageClass = 'ExportedGlobal';

Name the structure type in the generated code LUTStructType. Export the structure type definition to a generated header file named myLUTHdr.h.
LUTObj.StructTypeInfo.Name = 'LUTStructType';
LUTObj.StructTypeInfo.DataScope = 'Exported';
LUTObj.StructTypeInfo.HeaderFileName = 'myLUTHdr.h';

In an n-D Lookup Table block in a model, set **Data specification** to Lookup table object and **Name** to LUTObj.

`load_system('myModel_LUTObj')
set_param('myModel_LUTObj/Lookup Table','DataSpecification','Lookup table object',...
    'LookupTableObject','LUTObj')`

Generate code from the model.

`rtwbuild('myModel_LUTObj')`

### Starting build procedure for model: myModel_LUTObj
### Successful completion of code generation for model: myModel_LUTObj

The generated code defines the structure type **LUTStructType** in the generated header file **myLUTHdr.h**.

`file = fullfile('myModel_LUTObj_ert_rtw','myLUTHdr.h');
rtwdemodbtype(file,'typedef struct {','} LUTStructType;',1,1)`

```
typedef struct {
    int16_T myBPSet1[2];
    int16_T myBPSet2[5];
    int16_T myTable[10];
} LUTStructType;
```

The code uses the global structure variable **LUTObj** to store the table and breakpoint set data. The table data is scaled based on the specified fixed-point data type.

`file = fullfile('myModel_LUTObj_ert_rtw','myModel_LUTObj.c');
rtwdemodbtype(file,'LUTStructType LUTObj = {','/* Variable: LUTOбли */
LUTStructType LUTObj = { 
    { -1, 1 },
    { -2, -1, 0, 1, 2 },
    { 4, 29, 9, 34, 14, 39, 19, 44, 24, 49 } 
} ;         /* Variable: LUTObj`
Generate Code That Uses Conditionally Compiled Dimension Lengths

Suppose your handwritten code conditionally allocates memory and initializes a lookup table based on dimension lengths that you specify as `#define` macros. This example shows how to generate code that uses your external table and breakpoint data.

Symbolic dimensions require that you use an ERT-based system target file, which requires Embedded Coder®.

Explore External Code

In your current folder, copy these macro definitions into a header file named `ex_myHdr_LUT.h`.

```c
#include "rtwtypes.h"

#ifndef _HEADER_MYHDR_H_
define HEADER_MYHDR_H_

define bp1Len 2
#define bp2Len 2

typedef struct {
  real_T BP1[bp1Len];
  real_T BP2[bp2Len];
  real_T Table[bp1Len * bp2Len];
} LUTObj_Type;

extern LUTObj_Type LUTObj;
#endif

Copy this static initialization code into a source file named `ex_mySrc_LUT.c`.

```c
#include "ex_myHdr_LUT.h"

#if bp1Len == 2 && bp1Len == 2
LUTObj_Type LUTObj = {
  { 1.0, 2.0 },
  { 3.0, 4.0 },
};
#endif
```
To generate code that imports this data, create bp1Len and bp2Len as Simulink.Parameter objects in MATLAB. Create LUTObj as a Simulink.LookupTable object. Use the parameter objects to specify the dimension lengths for the table and breakpoint set data in the Simulink.LookupTable object.

Create Example Model

Create the example model ex_LUTObj by using an n-D Lookup Table block. In the Lookup Table block dialog box, on the Table and Breakpoints tab, set Number of table dimensions to 2.

open_system('ex_LUTObj')

Create Simulink.LookupTable Object

In the Model Explorer Model Hierarchy pane, select Base Workspace.

On the toolbar, click the Add Simulink LookupTable button. A Simulink.LookupTable object named Object appears in the base workspace.

In the Contents pane (the middle pane), rename the object as LUTObj.
Alternatively, create the object at the command prompt:

\[ \text{LUTObj} = \text{Simulink.LookupTable}; \]

**Configure Simulink.LookupTable Object**

In the **Contents** pane, select the new object LUTObj. The property dialog box appears in the **Dialog** pane (the right pane).

Set **Number of table dimensions** to 2.

Under **Table**, set **Value** to \([3 \ 4; \ 2 \ 1]\).

In the first row under **Breakpoints**, set **Value** to \([1 \ 2] \)

In the second row under **Breakpoints**, set **Value** to \([3 \ 4]\). Click **Apply**.

Under **Struct Type definition**, set **Data scope** to **Imported**. Set **Header file** to `ex_myHdr_LUT.h`. Set **Name** to LUTObj_Type.

In the Lookup Table block dialog box, set **Data specification** to **Lookup table object**. Set **Name** to LUTObj. Click **Apply**.

Alternatively, to configure the object and the blocks, use these commands:

```matlab
LUTObj.Breakpoints(1).Value = [1 2];
LUTObj.Breakpoints(2).Value = [3 4];
LUTObj.Table.Value = [3 4; 2 1];
LUTObj.StructTypeInfo.DataScope = 'Imported';
LUTObj.StructTypeInfo.HeaderFileName = 'ex_myHdr_LUT.h';
LUTObj.StructTypeInfo.Name = 'LUTObj_Type';
set_param('ex_LUTObj/Lookup Table','LookupTableObject','LUTObj')
set_param('ex_LUTObj/Lookup Table',...
    'DataSpecification','Lookup table object')
```

Enable the code generator to use Simulink.Parameter objects as macros that specify dimension lengths. Select the configuration parameter **Allow symbolic dimension specification**.

```matlab
set_param('ex_LUTObj','AllowSymbolicDim','on')
```

Create the Simulink.Parameter objects that represent the macros bp1Len and bp2Len. To generate code that imports the macros from your header file `ex_myHdr_LUT.h`, apply the storage class **ImportedDefine**.
bp1Len = Simulink.Parameter(2);
bp1Len.Min = 2;
bp1Len.Max = 3;
bp1Len.DataType = 'int32';
bp1Len.CoderInfo.StorageClass = 'Custom';
bp1Len.CoderInfo.CustomStorageClass = 'ImportedDefine';
bp1Len.CoderInfo.CustomAttributes.HeaderFile = 'ex_myHdr_LUT.h';

bp2Len = Simulink.Parameter(2);
bp2Len.Min = 2;
bp2Len.Max = 3;
bp2Len.DataType = 'int32';
bp2Len.CoderInfo.StorageClass = 'Custom';
bp2Len.CoderInfo.CustomStorageClass = 'ImportedDefine';
bp2Len.CoderInfo.CustomAttributes.HeaderFile = 'ex_myHdr_LUT.h';

Configure the existing Simulink.LookupTable object LUTOObj to use the Simulink.Parameter objects. Set the dimension lengths of the breakpoint set data and the table data by using the names of the parameter objects.

LUTOObj.Breakpoints(1).Dimensions = '[1 bp1Len]';
LUTOObj.Breakpoints(2).Dimensions = '[1 bp2Len]';
LUTOObj.Table.Dimensions = '[bp1Len bp2Len]';

Configure LUTOObj as imported data by applying the storage class ImportFromFile. To import your definition of LUTOObj, add the name of the file ex_mySrc_LUT.c to the model configuration parameter Configuration Parameters > Code Generation > Custom Code > Additional Build Information > Source files.

LUTOObj.CoderInfo.StorageClass = 'Custom';
LUTOObj.CoderInfo.CustomStorageClass = 'ImportFromFile';
LUTOObj.CoderInfo.CustomAttributes.HeaderFile = 'ex_myHdr_LUT.h';

set_param('ex_LUTOObj','CustomSource','ex_mySrc_LUT.c')

**Generate and Inspect Code**

Configure the model to compile an executable from the generated code.

set_param('ex_LUTOObj','GenCodeOnly','off')

Generate code from the model.

rtwbuild('ex_LUTOObj')
### Starting build procedure for model: ex_LUTObj
### Successful completion of build procedure for model: ex_LUTObj

In the code generation report, view the generated file ex_LUTObj.h. The file imports the macro definitions and the structure type definition by including your header file ex_myHdr_LUT.h.

```matlab
file = fullfile('ex_LUTObj_ert_rtw','ex_LUTObj.h');
rtwdemodbtype(file,'#include "ex_myHdr_LUT.h"','#include "ex_myHdr_LUT.h"',1,1)
```

```c
#include "ex_myHdr_LUT.h"
```

In the source file ex_LUTObj.c, the code algorithm in the model step function passes the breakpoint and table data to the function that performs the table lookup. The algorithm also passes bp1Len so the lookup function can traverse the rows and columns of the table data, which appear in the generated code as a serialized 1-D array.

```matlab
file = fullfile('ex_LUTObj_ert_rtw','ex_LUTObj.c');
rtwdemodbtype(file,'/* Model step function */','/* Model initialize function */',1,0)
```

```c
/* Model step function */
void ex_LUTObj_step(void) {
    /* Outport: '<Root>/Out1' incorporates:
    *  Inport: '<Root>/In1'
    *  Inport: '<Root>/In2'
    *  Lookup_n-D: '<Root>/Lookup Table'
    */
    ex_LUTObj_Y.Out1 = look2_binlcapw(ex_LUTObj_U.In1, ex_LUTObj_U.In2, 
    (&(LUTObj.BP1[0])), (&(LUTObj.BP2[0])), (&(LUTObj.Table[0])), 
    ex_LUTObj_ConstP.LookupTable_maxIndex, (uint32_T)bp1Len);
}
```

## Limitations

- You cannot subclass Simulink.Breakpoint or Simulink.LookupTable. For this reason, you cannot apply custom storage classes other than those in the built-in Simulink package.
- You cannot use Simulink.Breakpoint objects or Simulink.LookupTable objects that refer to Simulink.Breakpoint objects as instance-specific parameter data for reusable components. For example, you cannot use one of these objects as:

```c
/* Model step function */
void ex_LUTObj_step(void) {
    /* Outport: '<Root>/Out1' incorporates:
    *  Inport: '<Root>/In1'
    *  Inport: '<Root>/In2'
    *  Lookup_n-D: '<Root>/Lookup Table'
    */
    ex_LUTObj_Y.Out1 = look2_binlcapw(ex_LUTObj_U.In1, ex_LUTObj_U.In2, 
    (&(LUTObj.BP1[0])), (&(LUTObj.BP2[0])), (&(LUTObj.Table[0])), 
    ex_LUTObj_ConstP.LookupTable_maxIndex, (uint32_T)bp1Len);
}
```
• A model argument in a model workspace or a model argument value in a Model block.
• The value of a mask parameter on a CodeReuse Subsystem block.
• The value of a mask parameter on a subsystem that you reuse by creating a custom library.

However, you can use standalone Simulink.LoopupTable objects, which do not refer to Simulink.Breakpoint objects, in these ways.

• When blocks in a subsystem use Simulink.LoopupTable or Simulink.Breakpoint objects, you cannot set data type override only on the subsystem. Instead, set data type override on the entire model.

See Also

Topics
“Configure Lookup Tables for AUTOSAR Measurement and Calibration” (AUTOSAR Blockset)
“About Lookup Table Blocks”
“Package Shared Breakpoint and Table Data for Lookup Tables”
“Create Tunable Calibration Parameter in the Generated Code” (Simulink Coder)

Introduced in R2016b
Simulink.lookuptable.Breakpoint class

**Package:** Simulink.lookuptable

Configure breakpoint set data for lookup table object

**Description**

An object of the Simulink.lookuptable.Breakpoint class stores breakpoint set information for a lookup table. The object resides in the Breakpoints property of a Simulink.LookupTable object or Simulink.Breakpoint object.

You can use Simulink.LookupTable and Simulink.Breakpoint objects to store and configure a lookup table for ASAP2 and AUTOSAR code generation.

To represent multiple breakpoint sets for a multidimensional lookup table, store a vector of Simulink.lookuptable.Breakpoint objects in the Breakpoints property of a Simulink.LookupTable object.

To share a breakpoint set between multiple lookup tables, use a Simulink.Breakpoint object to store and configure the breakpoint set information. Use the object in a Prelookup block and create Simulink.LookupTable objects to use in Interpolation Using Prelookup blocks.

**Construction**

When you create a Simulink.LookupTable object or Simulink.Breakpoint object, a Simulink.lookuptable.Breakpoint object appears as the value of the Breakpoints property.

To create more Simulink.lookuptable.Breakpoint objects for a Simulink.LookupTable object, use this technique:

Access the Breakpoints property by specifying a vector index.

To create a Simulink.lookuptable.Breakpoint object, you can set the value of any of the object properties. The Simulink.LookupTable object creates the
Simulink.lookuptable.Breakpoint object with default property values, and sets the property that you specified.

The value of the Breakpoints property is an array of Simulink.lookuptable.Breakpoint objects. Each embedded object represents one breakpoint set.

For example, suppose that you create a Simulink.LookupTable object named LUTObj. To create more breakpoint sets, access the Breakpoints property by specifying vector indices:

```
LUTObj.Breakpoints(1).Value = [-1 1];
LUTObj.Breakpoints(2).Value = [-2 -1 0 1 2];
LUTObj.Breakpoints(3).Value = [-5 -3 0 3 5];
```

The object LUTObj creates additional Simulink.lookuptable.Breakpoint objects and sets the Value property of each object. LUTObj now stores information for three breakpoint sets.

**Properties**

**DataType — Data type of breakpoint set elements**

Character vector

Data type of breakpoint set elements, specified as a character vector. You can explicitly specify an integer, a floating-point, a fixed-point data type, or a data type expression such as the name of a Simulink.AliasType object.

The default value, 'auto', means that the breakpoint set acquires a data type from the value that you specify in the Value property. If you use an untyped expression such as [1 2 3] to set Value, the breakpoint data use the data type double. If you specify a typed expression such as single([1 2 3]) or an fi object, the breakpoint data use the data type specified by the expression or object.

For more information about data types in Simulink, see “Data Types Supported by Simulink”. To decide how to control the data types of table and breakpoint data in Simulink.LookupTable and Simulink.Breakpoint objects, see “Control Data Types of Lookup Table Objects” (SimulinkCoder).

Example: 'int16'

Example: 'myTypeAlias'
Data Types: char

**Description — Description of breakpoint set**
```
'' (default) | character vector
```

Description of the breakpoint set, specified as a character vector.
Example: 'This breakpoint set represents the pressure input.'
Data Types: char

**Dimensions — Dimension lengths of breakpoint set**
```
[0 0] (default) | numeric vector
```

Dimension lengths of the breakpoint set, returned as a numeric vector or specified as a character vector.
To use symbolic dimensions, specify a character vector.

**FieldName — Name of structure field that stores breakpoint set data**
```
'BP' (default) | character vector
```

Name of a structure field in the generated code, specified as a character vector. This field stores the breakpoint set data.

The code generator uses this property only under these circumstances, which cause the breakpoint data to appear in the generated code as a structure field:

- The `Simulink.lookuptable.Breakpoint` object exists in a `Simulink.LookupTable` object and in the `Simulink.LookupTable` object you set `BreakpointsSpecification` to 'Explicit values'.

Example: 'MyBkptSet1'
Data Types: char

**Max — Maximum value of breakpoint set elements**
```
[] (default) | numeric double value
```

Maximum value of the elements of the breakpoint set, specified as a numeric, real value of the data type double.
Example: 17.23
Data Types: double

**Min — Minimum value of breakpoint set elements**

Min value of the elements of the breakpoint set, specified as a numeric, real value of the data type double.

Example: -52.6
Data Types: double

**TunableSizeName — Name of structure field that stores length of breakpoint set**

Name of a structure field in the generated code, specified as a character vector. This field stores the length of the breakpoint set, which the generated code algorithm uses to determine the size of the table. To tune the effective size of the table during code execution, change the value of this structure field in memory.

The code generator uses this property only under these circumstances, which enable a tunable table size in the generated code:

- The `Simulink.lookuptable.Breakpoint` object exists in a `Simulink.LookupTable` object and in the `Simulink.LookupTable` object you set:
  - `BreakpointsSpecification` to 'Explicit values'.
  - `SupportTunableSize` to true.

Example: 'LengthForDim1'
Data Types: char

**Unit — Physical unit of breakpoint set**

Physical unit of the elements of the breakpoint set, specified as a character vector.

Example: 'inches'
Data Types: char

Value — Breakpoint set data
[] (default) | numeric vector

The breakpoint set data, specified as a numeric vector with at least two elements. To control the data type of the breakpoint set, use the DataType property of the Simulink.Breakpoint table.Breakpoint object.

When you set DataType to 'auto', to set the Value property, use a typed expression such as single([1 2 3]) or use the fi constructor to embed an fi object.

Example: [10 20 30]

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fi

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects (MATLAB).

See Also

Simulink.Breakpoint | Simulink.LookupTable

Introduced in R2016b
Simulink.lookuptable.Evenspacing class

Package: Simulink.lookuptable

Configure even spacing set data for lookup table object

Description

An object of the Simulink.lookuptable.Evenspacing class stores event spacing set information for a lookup table. The object resides in the Evenspacing property of a Simulink.LookupTable object.

You can use Simulink.LookupTable objects to store and configure a lookup table for ASAP2 and AUTOSAR code generation.

To represent multiple breakpoint sets for a multidimensional lookup table, store a vector of Simulink.lookuptable.Evenspacing objects in the Evenspacing property of a Simulink.LookupTable object.

Construction

When you create a Simulink.LookupTable object and set BreakpointSpecification to 'Even spacing', a Simulink.lookuptable.Evenspacing object appears as the value of the Breakpoints property.

To create more Simulink.lookuptable.Evenspacing objects for a Simulink.LookupTable object, use this technique:

Access the Breakpoints property by specifying a vector index.

To create a Simulink.lookuptable.Evenspacing object, you can set the value of any of the object properties. The Simulink.LookUpTable object creates the Simulink.lookuptable.Evenspacing object with default property values, and sets the property that you specified.
The value of the `Breakpoints` property is an array of `Simulink.lookuptable.Evenspacing` objects. Each embedded object represents one breakpoint set.

For example, suppose that you create a `Simulink.LookupTable` object named `myLUTObj`. To create more breakpoint sets, access the `Breakpoints` property by specifying scalar indices for `FirstPoint` and `Spacing` properties. To create more even spacing breakpoint sets, update the object with this pair of properties:

```plaintext
LUTObj.Breakpoints(1).FirstPoint=-1;
LUTObj.Breakpoints(1).Spacing=2;
LUTObj.Breakpoints(1).FirstPoint=-2;
LUTObj.Breakpoints(1).Spacing=1;
LUTObj.Breakpoints(1).FirstPoint=-5;
LUTObj.Breakpoints(1).Spacing=2;
```

The object `myLUTObj` creates additional `Simulink.lookuptable.Evenspacing` objects and sets the `FirstPoint` and `Spacing` properties of each object. `LUTObj` now stores information for three breakpoint sets.

### Properties

**FirstPoint — First point in evenly spaced breakpoint data**

[ ] (default) | numeric scalar

First point in evenly spaced breakpoint data, specified as a numeric scalar. To control the data type of the breakpoint set, use the `DataType` property of the `Simulink.lookuptable.Evenspacing` object.

When you set `DataType` to `'auto'`, to set the `FirstPoint` property, use a typed expression such as `single(1)` or use the `fi` constructor to embed a `fi` object.

Example: `-1`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fi`

**Spacing — Spacing between points in evenly spaced breakpoint data**

[ ] (default) | numeric, positive, monotonically increasing scalar

Spacing between points in evenly spaced breakpoint data, specified as a numeric scalar. To control the data type of the breakpoint set, use the `DataType` property of the `Simulink.lookuptable.Evenspacing` object.
When you set `DataType` to 'auto', to set the `Spacing` property, use a typed expression such as `single(1)` or use the `fi` constructor to embed an `fi` object.

Example: -1

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `fi`

### `DataType` — Data type of breakpoint set elements

`'auto'` (default) | character vector

Data type of breakpoint set elements, specified as a character vector. You can explicitly specify an integer, a floating-point, a fixed-point data type, or a data type expression such as the name of a `Simulink.AliasType` object.

The default value, 'auto', means that the breakpoint set acquires a data type from the value that you specify in the `Value` property. If you use an untyped expression such as `[1 2 3]` to set `Value`, the breakpoint data use the data type `double`. If you specify a typed expression such as `single([1 2 3])` or an `fi` object, the breakpoint data use the data type specified by the expression or object.

For more information about data types in Simulink, see “Data Types Supported by Simulink”. To decide how to control the data types of table and breakpoint data in `Simulink.LookupTable` and `Simulink.Breakpoint` objects, see “Control Data Types of Lookup Table Objects” (Simulink Coder).

Example: 'int16'

Example: 'myTypeAlias'

Data Types: `char`

### `Min` — Minimum value of breakpoint set elements

`[]` (default) | numeric double value

Minimum value of the elements of the breakpoint set, specified as a numeric, real value of the data type `double`.

Example: -52.6

Data Types: `double`

### `Max` — Maximum value of breakpoint set elements

`[]` (default) | numeric double value

Maximum value of the elements of the breakpoint set, specified as a numeric, real value of the data type `double`. 
Example: 17.23
Data Types: double

**Unit — Physical unit of breakpoint set**
'' (default) | character vector

Physical unit of the elements of the breakpoint set, specified as a character vector.
Example: 'inches'
Data Types: char

**FirstPointName — Name of the Simulink.lookuptable.Evenspacing object that stores the information for the first point**
'BPFIRSTPOINT1' (default) | character vector

Name of the Simulink.Breakpoint object that stores the information for the first point, specified as a character vector. Generated code uses this name to display the first point.
Example: 'myFirstPointName'
Data Types: char

**SpacingName — Name of the Simulink.lookuptable.Evenspacing object that stores the information for the spacing**
'auto' (default) | character vector

Name of the Simulink.Breakpoint object that stores the information for the spacing, specified as a character vector. Generated code uses this name to display the spacing.
Example: 'mySpacing'
Data Types: char

**TunableSizeName — Name of structure field that stores length of breakpoint set**
'N' (default) | character vector

Name of a structure field in the generated code, specified as a character vector. This field stores the length of the breakpoint set, which the generated code algorithm uses to determine the size of the table. To tune the effective size of the table during code execution, change the value of this structure field in memory.

The code generator uses the property only under these circumstances, which enable a tunable table size in the generated code:
The `Simulink.lookuptable.Evenspacing` object exists in a `Simulink.LookupTable` object, in which you set `BreakpointsSpecification` to 'Even spacing' and `SupportTunableSize` to true.

Example: 'LengthForDim1'

Data Types: char

Description — Description of breakpoint set

```
' ' (default) | character vector
```

Description of the breakpoint set, specified as a character vector.

Example: 'This breakpoint set represents the pressure input.'

Data Types: char

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects (MATLAB).

See Also

`Simulink.LookupTable` | `Simulink.lookuptable.Breakpoint`

Introduced in R2017b
Simulink.lookuptable.StructTypeInfo class

Package: Simulink.lookuptable

Configure settings for structure type that lookup table object uses in the generated code

Description

An object of the Simulink.lookuptable.StructTypeInfo class controls the structure type that the generated code creates to store data for lookup table objects. The Simulink.lookuptable.StructTypeInfo object resides in the StructTypeInfo property of a Simulink LookupTable object or Simulink Breakpoint object. Use these parent objects to store and configure a lookup table for ASAP2 and AUTOSAR code generation.

A Simulink LookupTable object appears as a structure in the generated code when you set the Specification property to 'Explicit values'. A Simulink Breakpoint object appears as a structure in the generated code when you set the SupportTunableSize property to true.

Construction

When you create a Simulink LookupTable or Simulink Breakpoint object, a Simulink.lookuptable.StructTypeInfo object appears as the value of the StructTypeInfo property.

Properties

DataScope — Scope of structure type definition

'StorageClass' | 'Auto' (default) | 'Exported' | 'Imported'

Scope of structure type definition, specified as a character vector.

Data Types: char
HeaderFileName — Name of header file that contains structure type definition
'' (default) | character vector

Name of the header file that contains the structure type definition, specified as a character vector.

Example: 'myHdr.h'

Data Types: char

Name — Name of structure type
'' (default) | character vector

Name of the structure type, specified as a character vector.

Data Types: char

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects (MATLAB).

See Also
Simulink.Breakpoint | Simulink.LookupTable

Introduced in R2016b
Simulink.lookuptable.Table class

Package: Simulink.lookuptable

Configure table data for lookup table object

Description

An object of the Simulink.lookuptable.Table class stores table information for a lookup table. The object resides in the Table property of a Simulink.LookupTable object. You can use the Simulink.LookupTable object to store and configure a lookup table for ASAP2 and AUTOSAR code generation.

Construction

When you create a Simulink.LookupTable object, a Simulink.lookuptable.Table object appears as the value of the Table property.

Properties

DataType — Data type of table data elements

'default' (default) | character vector

Data type of the table data elements, specified as a character vector. You can explicitly specify an integer data type, a floating-point data type, a fixed-point data type, or a data type expression such as the name of a Simulink.AliasType object.

The default value, 'auto', means that the table data acquire a data type from the value that you specify in the Value property. If you use an untyped expression such as [1 2 3] to set Value, the table data use the data type double. If you specify a typed expression such as single([1 2 3]) or an fi object, the table data use the data type specified by the expression or object.

For more information about data types in Simulink, see “Data Types Supported by Simulink”. To decide how to control the data types of table and breakpoint data in
Simulink.LookupTable and Simulink.Breakpoint objects, see “Control Data Types of Lookup Table Objects” (Simulink Coder).

Example: 'int16'
Example: 'myTypeAlias'
Data Types: char

Description — Description of table data
'' (default) | character vector

Description of the table data, specified as a character vector.
Example: 'This lookup table describes the action of a pump.'
Data Types: char

Dimensions — Dimension lengths of table data
[0 0] (default) | numeric vector

Dimension lengths of the table data, returned as a numeric vector or specified as a character vector.

To use symbolic dimensions, specify a character vector.

FieldName — Name of a structure field in the generated code
'Table' (default) | character vector

Name of a structure field in the generated code, specified as a character vector. This field stores the table data if you configure the Simulink.LookupTable object to appear in the generated code as a structure.

Example: 'MyPumpTable'
Data Types: char

Max — Maximum value of table data elements
[] (default) | numeric double value

Maximum value of the elements of the table data, specified as a numeric, real value of the data type double.
Example: 17.23
Data Types: double
Min — Minimum value of table data elements

[[] (default) | numeric double value

Minimum value of the elements of the table data, specified as a numeric, real value of the data type double.

Example: -52.6

Data Types: double

Unit — Physical unit of table elements

' ' (default) | character vector

Physical unit of the elements of the table data, specified as a character vector.

Example: 'degC'

Data Types: char

Value — Table data

[[] (default) | numeric vector or multidimensional array

The table data, specified as a numeric vector or multidimensional array with at least two elements. To control the data type of the table data, use the DataType property of the Simulink.lookuptable.Table object.

When you set DataType to 'auto', to set the Value property, use a typed expression such as single([1 2 3]) or use the fi constructor to embed an fi object.

Example: [10 20 30; 40 50 60]

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | fi

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects (MATLAB).

See Also

Simulink.Breakpoint | Simulink.LookupTable

Introduced in R2016b
Simulink.MDLInfo class

Package: Simulink

Extract model file information without loading block diagram into memory

Description

The class Simulink.MDLInfo extracts information from a model file without loading the block diagram into memory.

You can create an MdlInfo object containing all the model information properties, or you can use the static methods for convenient access to individual properties without creating the class first. For example, to get the description only:

description = Simulink.MDLInfo.getDescription('mymodel')

To get the metadata only:

metadata = Simulink.MDLInfo.getMetadata('mymodel')

All model information properties are read only.

Construction

info = Simulink.MDLInfo('mymodel') creates an instance of the MdlInfo class info and populates the properties with the information from the model file 'mymodel'.

mymodel can be:

- A block diagram name (for example, vdp)
- The file name for a file on the MATLAB path (for example, mymodel.slx)
- A file name relative to the current folder (for example, mydir/mymodel.slx)
- A fully qualified file name (for example, C:\mydir\mymodel.slx)

Simulink.MDLInfo resolves the supplied name by looking at files on the MATLAB path, and ignores any block diagrams in memory. This may cause unexpected results if you
supply the name of a loaded model, but its file is shadowed by another file on the MATLAB path. If a file is shadowed, you see a warning in the command window. To avoid any confusion, supply a fully-qualified file name to Simulink.MDLInfo.

**Properties**

**BlockDiagramName**

Name of block diagram.

**BlockDiagramType**

Type of block diagram. For example, 'Model', Subsystem', 'Library'.

**Description**

The Description parameter of the model. For details, see “Version Information Properties”.

**FileName**

The fully-qualified Name of the model file.

**Interface**

Names and attributes of the block diagram's root inports, outports, model references, etc., describing the graphical interface if you created a Model Reference block from this model.

Structure.

**IsLibrary**

Whether the block diagram is a library. Logical.
**LastModifiedBy**

Name of the user who last saved the model.

**LastSavedArchitecture**

Platform architecture when saved, for example, 'glxnx64'.

**Metadata**

Names and values of arbitrary data associated with the model.

Structure. The structure fields can be character vectors, numeric matrices of type "double", or more structures. Use the method `getMetadata` to extract this metadata structure without loading the model.

**ModelVersion**

Model file version number.

**ReleaseName**

Name of release of Simulink that was used to save the model file, for example, 'R2016a'.

**SavedCharacterEncoding**

Character encoding when saved, for example, 'UTF-8'.

**SimulinkVersion**

Version number of Simulink software that was used to save the model file.

**Methods**

getDescription Extract model file description without loading block diagram into memory

getMetadata Extract model file metadata without loading block diagram into memory
Copy Semantics

Handle. To learn how this affects your use of the class, see Copying Objects (MATLAB) in the MATLAB Programming Fundamentals documentation.

Examples

Construct and view a model information object:

```matlab
info = Simulink.MDLInfo('mymodel')
% Get the Version when the model was saved
simulink_version = info.SimulinkVersion;
% Get model metadata
metadata = info.metadata
```

To add metadata to a model, create a metadata structure containing the information you require and use `set_param` to attach it to the model. For example:

```matlab
metadata.TestStatus = 'untested';
metadata.ExpectedCompletionDate = '01/01/2011';
load_system('mymodelname');
set_param('mymodelname','Metadata',{}
metadata) % must be a struct
save_system('mymodelname');
close_system('mymodelname');
```

Construct a model information object for a model named `mpowertrain`, in order to find the names of referenced models without loading the model into memory:

```matlab
info = Simulink.MDLInfo('mpowertrain')
% Get the Interface property
info.Interface
```

Output:

```matlab
ans =
    Inports: [0x1 struct]
    Outports: [0x1 struct]
    Trigports: [0x1 struct]
    Connports: [0x1 struct]
    ModelVersion: '1.122'
```
Get the referenced models:

```
info.Interface.ModelReferences
```

Output is in the form `model name / block path | referenced model name`:

```
ans =
    'mpowertrain/Model Variants|manual_transmission'
    'mpowertrain/engine model|menginemodel'
```

**See Also**

Simulink.MDLInfo.getDescription; Simulink.MDLInfo.getMetadata
Simulink.MDLInfo.getDescription

Class: Simulink.MDLInfo
Package: Simulink

Extract model file description without loading block diagram into memory

Syntax

description = Simulink.MDLInfo.getDescription('mymodel')
description = info.getDescription

Description

description = Simulink.MDLInfo.getDescription('mymodel') returns the description associated with the file mymodel, without loading the model.

mymodel can be:

• A block diagram name (for example, vdp)
• The file name for a file on the MATLAB path (for example, mymodel.slx)
• A file name relative to the current folder (for example, mydir/mymodel.slx)
• A fully qualified file name (for example, C:\mydir\mymodel.slx)

description = info.getDescription returns the description property of the Simulink.MDLInfo object info.

Examples

Get the description without loading the model or creating a Simulink.MDLInfo object:

description = Simulink.MDLInfo.getDescription('mymodel')

Create a Simulink.MDLInfo object containing all the model information properties, and get the description property:
info = Simulink.MDLInfo('mymodel')
description = info.getDescription

See Also

Simulink.MDLInfo; Simulink.MDLInfo.getMetadata
Simulink.MDLInfo.getMetadata

Class: Simulink.MDLInfo  
Package: Simulink

Extract model file metadata without loading block diagram into memory

Syntax

metadata = Simulink.MDLInfo.getMetadata('mymodel')
metadata = info.getMetadata

Description

metadata = Simulink.MDLInfo.getMetadata('mymodel') extracts the structure metadata associated with the file mymodel, without loading the model.

mymodel can be:

- A block diagram name (for example, vdp)
- The file name for a file on the MATLAB path (for example, mymodel.slx)
- A file name relative to the current folder (for example, mydir/mymodel.slx)
- A fully qualified file name (for example, C:\mydir\mymodel.slx)

metadata = info.getMetadata returns the metadata property of the Simulink.MDLInfo object info.

metadata is a structure containing the names and attributes of arbitrary data associated with the model. The structure fields can be character vectors, numeric matrices of type "double", or more structures.

To add metadata to a model, create a metadata structure containing the information you require and use set_param to attach it to the model. If it is important to extract the information without loading the model, use metadata instead of adding custom user data with add_param.
Examples

Create a metadata structure and use `set_param` to attach it to the model:

```matlab
metadata.TestStatus = 'untested';
metadata.ExpectedCompletionDate = '01/01/2011';
load_system('mymodel');
set_param('mymodel','Metadata',metadata) % must be a struct
save_system('mymodel');
close_system('mymodel');
```

Get the metadata without loading the model or creating a `Simulink.MDLInfo` object:

```matlab
metadata = Simulink.MDLInfo.getMetadata('mymodel')
```

Create a `Simulink.MDLInfo` object containing all the model information properties, and get the metadata property:

```matlab
info = Simulink.MDLInfo('mymodel')
metadata = info.Metadata
```

See Also

`Simulink.MDLInfo`; `Simulink.MDLInfo.getDescription`
Simulink.ModelAdvisor

Run Model Advisor from MATLAB file

Description

To run the Model Advisor, use instances of this class in MATLAB programs. For example, you can perform a standard set of checks.

Many Simulink.ModelAdvisor object methods require or return IDs. An ID is a unique identifier for a Model Advisor check, task, or group. IDs must remain constant. A Simulink.ModelAdvisor object includes methods that enable you to retrieve the ID or IDs for:

- All checks, tasks, and groups
- Checks belonging to groups and tasks
- The active check
- Selected checks, tasks, and groups

To find check IDs in the Model Advisor, use the check context menu.

<table>
<thead>
<tr>
<th>Find</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Check Title, ID, or location of the MATLAB source code</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>On the model window toolbar, select <strong>Settings &gt; Preferences</strong>.</td>
</tr>
<tr>
<td>2</td>
<td>In the Model Advisor Preferences dialog box, select <strong>Show Source Tab</strong>.</td>
</tr>
<tr>
<td>3</td>
<td>In the right pane of the Model Advisor window, click the <strong>Source</strong> tab. The Model Advisor window displays the check Title, TitleId, and location of the MATLAB source code for the check.</td>
</tr>
</tbody>
</table>

| Check ID | 
| 1 | In the left pane of the Model Advisor, select the check. |
| 2 | Right-click the check name and select **Send Check ID to Workspace**. The ID is displayed in the Command Window and sent to the base workspace. |
Find | Action
---|---
Check IDs for selected checks in a folder | 1. In the left pane of the Model Advisor, select the checks for which you want IDs. Clear the other checks in the folder.
2. Right-click the folder and select **Send Check ID to Workspace**. An array of the selected check IDs are sent to the base workspace.

**Creation**

**Syntax**

```matlab
ma = Simulink.ModelAdvisor.getModelAdvisor(system)
```

**Description**

MATLAB creates an instance of this object for each model that you open in the current MATLAB session. To get a handle to a model's Model Advisor object, execute this command:

```matlab
ma = Simulink.ModelAdvisor.getModelAdvisor(system)
```

where `system` is the name of the model or subsystem that you want to check. Your program can then use the Model Advisor object methods to initialize and run the Model Advisor checks.

**Input Arguments**

- **system** — Model or subsystem name
  
  Character vector

  Model or subsystem that the Model Advisor checks.

  Data Types: char
Properties

EmitInputParametersToReport — Display check input parameters in Model Advisor report
true (default) | false

A value of true displays input parameters in the Model Advisor report. A value of false does not display input parameters in the Model Advisor report.

Data Types: char

Object Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>closeReport</td>
<td>Close Model Advisor report</td>
</tr>
<tr>
<td>deselectCheck</td>
<td>Clear Model Advisor check</td>
</tr>
<tr>
<td>deselectCheckAll</td>
<td>Clear Model Advisor checks</td>
</tr>
<tr>
<td>deselectCheckForGroup</td>
<td>Clear Model Advisor checks for groups</td>
</tr>
<tr>
<td>deselectCheckForTask</td>
<td>Clear Model Advisor checks that belong to a specified task or set of tasks</td>
</tr>
<tr>
<td>deselectTask</td>
<td>Clear Model Advisor checks that belong to a specified task or set of tasks</td>
</tr>
<tr>
<td>deselectTaskAll</td>
<td>Clear all Model Advisor tasks</td>
</tr>
<tr>
<td>displayReport</td>
<td>Display report in Model Advisor window</td>
</tr>
<tr>
<td>exportReport</td>
<td>Create copy of Model Advisor report</td>
</tr>
<tr>
<td>filterResultWithExclusion</td>
<td>Filter excluded objects</td>
</tr>
<tr>
<td>getBaselineMode</td>
<td>Determine whether Model Advisor is in baseline data generation mode</td>
</tr>
<tr>
<td>getCheckAll</td>
<td>Get check IDs of all Model Advisor checks</td>
</tr>
<tr>
<td>getCheckForGroup</td>
<td>Get checks belonging to Model Advisor check group</td>
</tr>
<tr>
<td>getCheckForTask</td>
<td>Get checks belonging to a task</td>
</tr>
<tr>
<td>getCheckResult</td>
<td>Get Model Advisor check results</td>
</tr>
<tr>
<td>getCheckResultData</td>
<td>Obtain Model Advisor check result data</td>
</tr>
<tr>
<td>getCheckResultStatus</td>
<td>Obtain Model Advisor check result status</td>
</tr>
<tr>
<td>getGroupAll</td>
<td>Get all groups of checks that Model Advisor runs</td>
</tr>
<tr>
<td>getInputParameters</td>
<td>Get Model Advisor check input parameters</td>
</tr>
<tr>
<td>getListViewParameters</td>
<td>Get Model Advisor checklist parameters displayed in checklist view</td>
</tr>
</tbody>
</table>
Simulink.ModelAdvisor.getModelAdvisor

getSelectedCheck
getSelectedSystem
getSelectedTask
getTaskAll
Simulink.ModelAdvisor.openConfigUI
Simulink.ModelAdvisor.reportExists

runCheck
runTask
selectCheck
selectCheckAll
selectCheckForGroup
selectCheckForTask
selectTask
selectTaskAll
setActionEnable
setBaselineMode
setCheckErrorSeverity
setCheckResult
setCheckResultData
setListViewParameters
setInputParameters
verifyCheckRan
verifyCheckResult

verifyCheckResultStatus

verifyHTML

See Also

Topics
“Run Model Advisor Checks”
Introduced in R2006a
closeReport

Package: Simulink

Close Model Advisor report

Syntax

closeReport(ma)

Description

For a Model Advisor object, use the closeReport(ma) method to close a Model Advisor Report.

Input Arguments

ma — Model Advisor object
Simulink.ModelAdvisor object

Simulink.ModelAdvisor object for which you want to close the corresponding Model Advisor report.

See Also

Simulink.ModelAdvisor

Introduced in R2006a
deselectCheck

Package: Simulink

Clear Model Advisor check

Syntax

output = deselectCheck(ma,checkID)

Description

For a Model Advisor object, use the `output = deselectCheck(ma,checkID)` method to clear a Model Advisor check. This method cannot clear disabled checks.

Input Arguments

- **ma** — Model Advisor object
  Simulink.ModelAdvisor object
  Simulink.ModelAdvisor object that you want to clear Model Advisor check(s).

- **checkID** — Check IDs associated with Model Advisor check(s)
  character vector | cell array of character vectors
  Model Advisor Check IDs for which you want to clear from the Model Advisor.
  Example: 'mathworks.design.UnconnectedLinesPorts'

Output Arguments

- **output** — Logical value
  boolean
A logical value indicating whether Simulink cleared the specified check. A value of 1 indicates success. A value of 0 indicates that Simulink was not able to clear the specified check.

See Also
Simulink.ModelAdvisor

Introduced in R2006a
deselectCheckAll

Package: Simulink

Clear Model Advisor checks

Syntax

output = deselectCheckAll(ma)

Description

For a Model Advisor object, use the `output = deselectCheckAll(ma)` method to clear all selected Model Advisor checks. This method cannot clear disabled checks.

Input Arguments

`ma` — Model Advisor object
Simulink.ModelAdvisor object

Simulink.ModelAdvisor object for which you want to clear all selected Model Advisor checks.

Output Arguments

`output` — Logical value
boolean

A logical value indicating whether Simulink cleared all selected checks. A value of 1 indicates success. A value of 0 indicates that Simulink did not clear all checks.

See Also

Simulink.ModelAdvisor
Introduced in R2006a
deselectCheckForGroup

Package: Simulink

Clear Model Advisor checks for groups

Syntax

output = deselectCheckForGroup(ma,groupName)

Description

For a Model Advisor object, use the output = deselectCheckForGroup(ma,groupName) method to clear all selected Model Advisor checks for one or more groups. To locate the group name, use the getGroupAll method.

Input Arguments

ma — Model Advisor object
Simulink.ModelAdvisor object

Simulink.ModelAdvisor object for which you want to clear all selected Model Advisor checks for one or more groups.

groupName — Check group names
character vector | cell array of character vectors

Name of group or groups that contain the selected Model Advisor checks that you want to clear.

Output Arguments

output — Logical value
boolean

5-326
A logical value indicating whether Simulink cleared the checks for the specified group. A value of 1 indicates success. A value of 0 indicates that Simulink did not clear the specified checks.

**See Also**
Simulink.ModelAdvisor

**Introduced in R2006a**
deselectCheckForTask

**Package:** Simulink

Clear Model Advisor checks that belong to a specified task or set of tasks

**Syntax**

```matlab
output = deselectCheckForTask(ma,task)
```

**Description**

For a Model Advisor object, use the `output = deselectCheckForTask(ma,task)` method to clear all selected Model Advisor checks for one or more tasks. To locate the task name, use the `getTaskAll` method.

**Input Arguments**

- **`ma` — Model Advisor object**
  - `Simulink.ModelAdvisor` object
  - Simulink `ModelAdvisor` object for which you want to clear all selected Model Advisor checks for a task or tasks

- **`task` — Task name**
  - `character vector` | `cell array of character vectors`
  - Name of task or tasks that contain selected Model Advisor checks that you want to clear.

**Output Arguments**

- **`output` — Logical value**
  - `boolean`

---

5-328
A logical value indicating whether Simulink cleared the checks for the specified group. A value of 1 indicates success. A value of 0 indicates that Simulink did not clear the specified checks.

**See Also**
Simulink.ModelAdvisor

**Introduced in R2006a**
deselectTask

**Package:** Simulink

Clear Model Advisor checks that belong to a specified task or set of tasks

**Syntax**

```plaintext
output = deselectTask(ma,task)
```

**Description**

For a Model Advisor object, use the `output = deselectTask(ma,task)` method to clear all selected Model Advisor checks for one or more tasks. To locate the task name, use the `getTaskAll` method.

**Input Arguments**

- `ma` — Model Advisor object
  
  Simulink.ModelAdvisor object

  Simulink.ModelAdvisor object for which you want to clear all selected Model Advisor checks for a task or tasks

- `task` — Task name
  
  character vector | cell array of character vectors

  Name of task or tasks that contain selected Model Advisor checks that you want to clear.

**Output Arguments**

- `output` — Logical value
  
  boolean

  Whether the selected checks were cleared as a boolean value.
A logical value indicating whether Simulink removed the checks for the specified task. A value of 1 indicates success. A value of 0 indicates that Simulink was not able to clear the checks for the specified task.

**See Also**
Simulink.ModelAdvisor

**Introduced in R2006a**
**deselectTaskAll**

**Package:** Simulink

Clear all Model Advisor tasks

**Syntax**

```matlab
output = deselectTaskAll(ma)
```

**Description**

For a Model Advisor object, use the `output = deselectTaskAll(ma)` method to clear all selected Model Advisor checks for all tasks.

**Input Arguments**

`ma` — Model Advisor object

`Simulink.ModelAdvisor` object

`Simulink.ModelAdvisor` object for which you want to clear all selected Model Advisor checks for all tasks

**Output Arguments**

`output` — Logical value

`boolean`

A logical value indicating whether Simulink cleared the checks for all tasks. A value of 1 indicates success. A value of 0 indicates that Simulink did not clear the checks.

**See Also**

`Simulink.ModelAdvisor`
Introduced in R2006a
displayReport

Package: Simulink

Display report in Model Advisor window

Syntax

displayReport(ma)

Description

For a Model Advisor object, use the displayReport(ma) method to display a Model Advisor report in the Model Advisor window. The report includes the most recent results of running checks on the system corresponding to the Model Advisor object and the current selection status of checks, groups, and tasks for the system.

Input Arguments

ma — Model Advisor object
Simulink.ModelAdvisor object

Simulink.ModelAdvisor object for which you want to open the corresponding Model Advisor report.

See Also

Simulink.ModelAdvisor

Introduced in R2006a
**exportReport**

**Package:** Simulink

Create copy of Model Advisor report

**Syntax**

```matlab
[output, message] = exportReport(ma, destination)
```

**Description**

For a Model Advisor object, use the `exportReport(ma, destination)` method to create a copy of a Model Advisor report. The report includes the most recent results of running checks on the system corresponding to the Model Advisor object and the current selection status of checks, groups, and tasks for the system.

**Input Arguments**

- **ma** — Model Advisor object
  - `Simulink.ModelAdvisor` object
  - `Simulink.ModelAdvisor` object for which you want to export the corresponding Model Advisor report.

- **destination** — Path name
  - `character vector`
  - Path name to report file location.

**Output Arguments**

- **output** — Logical value
  - `boolean`

A logical value indicating whether Simulink created a report in location that you specify by destination input argument. A value of 1 indicates success. A value of 0 indicates that Simulink did not create a copy of the report in the specified location.

message — Reason why report was not created
character vector

This output argument is empty if a report is created. If a report was not created, this argument contains the reason why.

See Also
Simulink.ModelAdvisor

Introduced in R2006a
filterResultWithExclusion

**Package:** Simulink

Filter excluded objects

**Syntax**

```
filteredResultHandles = filterResultWithExclusion(ma, ResultHandles)
```

**Description**

Use the `filteredResultHandles = filterResultWithExclusion(ma, ResultHandles)` method to filter objects that have exclusions enabled and cause a check warning or failure.

This method is intended for excluding objects from custom checks that you create by using the Model Advisor customization API, available with Simulink Check™.

**Input Arguments**

- **ma — Model Advisor object**
  
  `Simulink.ModelAdvisor` object

  Simulink.ModelAdvisor object for which you want to filter objects causing exclusion enabled checks to warn or fail.

- **ResultHandles — Array of model objects causing a check warning or failure**
  
  Array of model objects

  Array of objects causing a check warning or failure.
Output Arguments

filteredResultHandles — Array of objects causing a check warning or failure
array of objects

An array of objects causing exclusion enabled checks to warn or fail.

See Also
Simulink.ModelAdvisor

Topics
“Exclude Blocks From Custom Checks” (Simulink Check)

Introduced in R2006a
getBaselineMode

Package: Simulink

Determine whether Model Advisor is in baseline data generation mode

Syntax

mode = getBaselineMode(ma)

Description

mode = getBaselineMode(ma) returns true if the Model Advisor is in baseline data mode. Baseline data mode causes the verification methods of the Model Advisor, for example, verifyHTML, to generate baseline data.

Input Arguments

ma — Model Advisor object
Simulink.ModelAdvisor object

Simulink.ModelAdvisor object for which you want to determine if it is in baseline data mode.

Output Arguments

mode — Boolean value indicating baseline mode
0 (default)

A value of 1 indicates that the Model Advisor is in baseline data generation mode.

See Also

Simulink.ModelAdvisor
Introduced in R2006a
**getCheckAll**

**Package:** Simulink

Get check IDs of all Model Advisor checks

**Syntax**

```matlab
CheckIDs = getCheckAll(ma)
```

**Description**

The `CheckIDs = getCheckAll(ma)` method returns a cell array of character vectors specifying the Check IDs of all checks that the Model Advisor performs.

**Input Arguments**

- `ma` — Model Advisor object
  
  *Simulink.ModelAdvisor* object
  
  *Simulink.ModelAdvisor* object for which you want to get the Check IDs of all checks that the Model Advisor performs.

**Output Arguments**

- `CheckIDs` — Model Advisor Check IDs
  
  *cell array of character vectors*
  
  Cell array of character vectors specifying the Check IDs of all Model Advisor checks.

**See Also**

`Simulink.ModelAdvisor`
Introduced in R2006a
getCheckForGroup

Package: Simulink

Get checks belonging to Model Advisor check group

Syntax

groupChecks = getCheckForGroup(ma,groupName)

Description

For a Model Advisor object, use the groupChecks = getCheckForGroup(ma, groupName) method to get all of the Model Advisor checks in a group. To locate the group name, use the getGroupAll method.

Input Arguments

ma — Model Advisor object
Simulink.ModelAdvisor object

Simulink.ModelAdvisor object for which you want to get all of the Model Advisor checks corresponding to a check group.

groupName — Check group name
character vector

Name of group that contains the Model Advisor checks.

Example: 'Simulink Coder'

Output Arguments

groupChecks — Model Advisor checks

cell array of character vectors
Checks that belong to a Model Advisor group.

See Also
Simulink.ModelAdvisor

Introduced in R2006a
getCheckForTask

**Package:** Simulink

Get checks belonging to a task

**Syntax**

```matlab
taskChecks = getCheckForTask(ma, taskID)
```

**Description**

For a Model Advisor object, use the `taskChecks = getCheckForTask(ma, taskID)` method to get all of the Model Advisor checks that are part of a task. To locate the task name, use the `getTaskAll` method.

**Input Arguments**

*ma* — **Model Advisor object**

`Simulink.ModelAdvisor` object

Simulink `ModelAdvisor` object for which you want to get all of the Model Advisor checks that are part of a task.

*taskID* — **Task name**

`character vector`

Name of task that contains the Model Advisor checks.

Example: `'_SYSTEM_By Task_Modeling_Physical_Systems'`

**Output Arguments**

*taskChecks* — **Model Advisor checks**

`cell array of character vectors`
Checks that are part of a Model Advisor task

**See Also**
Simulink.ModelAdvisor

**Introduced in R2006a**
getCheckResult

Package: Simulink

Get Model Advisor check results

Syntax

results = getCheckResult(ma,checkID)

Description

For a Model Advisor object, use the results = getCheckResult(ma,checkID) method to get the results of running a check or set of checks.

This method enables you to access results generated by custom checks that you create using the Model Advisor customization API, an optional feature available with Simulink Check software. For more information, see “Define Custom Checks” (Simulink Check).

Input Arguments

ma — Model Advisor object
Simulink.ModelAdvisor object

Simulink.ModelAdvisor object for which you want to obtain check results.

checkID — Check IDs associated with Model Advisor check
cell array of character vectors

Model Advisor Check IDs for which you want to obtain results.
Example: 'mathworks.design.UnconnectedLinesPorts'
Output Arguments

results — Check results

cell array

Cell array of check results. The result format depends on the check data.

See Also

Simulink.ModelAdvisor

Introduced in R2006a
**getCheckResultData**

**Package:** Simulink

Obtain Model Advisor check result data

**Syntax**

```matlab
results = getCheckResultData(ma,checkID)
```

**Description**

For a Model Advisor object, use the `results = getCheckResultData(ma,checkID)` method to get the check result data for a specified set of checks.

This method enables you to access results generated by custom checks that you create using the Model Advisor customization API, an optional feature available with Simulink Check software. For more information, see “Define Custom Checks” (Simulink Check).

**Input Arguments**

- **ma** — Model Advisor object
  
  *Simulink.ModelAdvisor* object

  Simulink.ModelAdvisor object for which you want to obtain check result data.

- **checkID** — Check IDs associated with Model Advisor checks
  
  *character vector | cell array of character vectors*

  Model Advisor Check IDs that you want to obtain results

  Example: `'mathworks.design.UnconnectedLinesPorts'`
Output Arguments

results — Check results
cell array

Data from a check result or cell array of data from check results. The data format depends on the checks that generate the data.

See Also
Simulink.ModelAdvisor

Introduced in R2006a
getCheckResultStatus

Package: Simulink

Obtain Model Advisor check result status

Syntax

status = getCheckResultStatus(ma, checkID)

Description

After running a set of checks, use the `status = getCheckResultStatus(ma, checkID)` method to determine whether these checks passed or failed.

Input Arguments

ma — Model Advisor object
Simulink.ModelAdvisor object

Simulink.ModelAdvisor object for which you want to determine whether a check or set of checks passed or failed.

cHECKID — Model Advisor Check IDs
character vector | cell array of character vectors

Model Advisor Check IDs for which you want to obtain results.
Example: 'mathworks.design.UnconnectedLinesPorts'

Output Arguments

status — Pass or fail
Boolean | cell array of Boolean values
Boolean value or cell array of Boolean values indicating the pass or fail status of a check or set of checks.

See Also
Simulink.ModelAdvisor

Introduced in R2006a
getGroupAll

Package: Simulink

Get all groups of checks that Model Advisor runs

Syntax

groupChecks = getGroupAll(ma)

Description

For a Model Advisor object, use the `groupChecks = getGroupAll(ma)` method to get all groups of checks that the Model Advisor runs.

Input Arguments

ma — Model Advisor object
Simulink.ModelAdvisor object

Simulink.ModelAdvisor object for which you want to determine all Model Advisor check groups.

Output Arguments

groupChecks — Model Advisor check groups
cell array of character vectors

IDs of check groups that the Model Advisor runs.

See Also

Simulink.ModelAdvisor
Introduced in R2006a
**getInputParameters**

**Package:** Simulink

Get Model Advisor check input parameters

**Syntax**

```matlab
params = getInputParameters(ma,CheckID)
```

**Description**

Use the `params = getInputParameters(ma,CheckID)` method to get the input parameters for a Model Advisor check.

This method enables you to access custom checks that you create using Simulink Check. For more information, see “Define Custom Checks” (Simulink Check).

**Input Arguments**

- **ma — Model Advisor object**
  
  Simulink.ModelAdvisor object

  `Simulink.ModelAdvisor` object for which you determine the input parameters for specific Model Advisor check.

- **CheckID — Model Advisor check**
  
  character vector

  Model Advisor check ID for which to obtain its input parameters. If you use the method inside a check callback function, you can omit this argument.

  Example: `‘mathworks.design.UnconnectedLinesPorts’`
Output Arguments

params — Input parameters
cell array of ModelAdvisor.InputParameter objects

Cell array that contains the ModelAdvisor.InputParameter objects for a Model Advisor check.

See Also
Simulink.ModelAdvisor

Topics
“Run Model Advisor Checks”

Introduced in R2006a
**getListViewParameters**

**Package:** Simulink

Get Model Advisor checklist parameters displayed in checklist view

**Syntax**

```matlab
params = getListViewParameters(ma,CheckID)
```

**Description**

Use the `params = getListViewParameters(ma,CheckID)` method to get the list view of parameters for a Model Advisor check.

This method enables you to access custom checks that you create by using Simulink Check. For more information, see “Define Custom Checks” (Simulink Check).

**Input Arguments**

- `ma` — **Model Advisor object**  
  `Simulink.ModelAdvisor` object

  Simulink.ModelAdvisor object for which you determine the list view of parameters for a specific Model Advisor check.

- `CheckID` — **Model Advisor check**  
  `character vector`

  Name of Model Advisor check for which to obtain its parameters in list view. You can omit this argument if you use the method inside a check callback function.

  Example: `mathworks.design.UnconnectedLinesPorts`
Output Arguments

```plaintext
params — Input parameters
 cell array of ModelAdvisor.ListViewParameter objects
```

Cell array that contains the `ModelAdvisor.ListViewParameter` objects for a Model Advisor check.

See Also

- `ModelAdvisor.ListViewParameter` | `Simulink.ModelAdvisor`

Introduced in R2006a
Simulink.ModelAdvisor.getModelAdvisor

Package: Simulink

Get Model Advisor object for system or subsystem

Syntax

ma = Simulink.ModelAdvisor.getModelAdvisor(system,new)

Description

Use the `ma = Simulink.ModelAdvisor.getModelAdvisor(system,new)` method to return an instance of a `Simulink.ModelAdvisor` class for the model or subsystem that you specify by using the `system` input argument.

Input Arguments

system — Model or subsystem name
character vector

Name of model or subsystem for which you obtain the corresponding `Simulink.ModelAdvisor` object.

Example: 'sldemo_fuelsys_dd_controller'
Example: 'sldemo_fuelsys_dd_controller/airflow_calc'

Data Types: char

new — New Model Advisor object
character vector

This argument is optional except when changing the Model Advisor working scope from one system to another without closing the previous session. Alternatively, you can close the previous session before invoking the `getModelAdvisor` method.

Data Types: char
Output Arguments

ma — Model Advisor object
Simulink.ModelAdvisor object

Simulink.ModelAdvisor object for a model or subsystem that you specify by using a system input argument.

See Also
Simulink.ModelAdvisor

Introduced in R2006a
getSelectedCheck

Package: Simulink

Get currently selected Model Advisor checks

Syntax

CheckIDs = getSelectedCheck(ma)

Description

Use the `CheckIDs = getSelectedCheck(ma)` method to obtain the selected Model Advisor checks for a corresponding `Simulink.ModelAdvisor` object.

Input Arguments

`ma` — Model Advisor object
 `Simulink.ModelAdvisor` object

`Simulink.ModelAdvisor` object for which you want to obtain the selected Model Advisor checks.

Output Arguments

`CheckIDs` — Model Advisor checks
cell array of character vectors

Check IDs of currently selected Model Advisor checks.

See Also

`Simulink.ModelAdvisor`
Introduced in R2006a
getSelectedSystem

Package: Simulink

Get system that Model Advisor targets

Syntax

system = getSelectedSystem(ma)

Description

Use the system = getSelectedSystem(ma) method to obtain the model or subsystem that is the target of the Simulink.ModelAdvisor object.

Input Arguments

ma — Model Advisor object
Simulink.ModelAdvisor object

Simulink.ModelAdvisor object for which you want to obtain the model or subsystem that the object targets.

Output Arguments

system — Model or subsystem
character vector

Name of the model or subsystem that the Model.Advisor object ma targets.

See Also

Simulink.ModelAdvisor
Introduced in R2006a
getSelectedTask

Package: Simulink

Get selected Model Advisor tasks

Syntax

TaskIDs = getSelectedTask(ma)

Description

Use the TaskIDs = getSelectedTask(ma) method to obtain the tasks that are selected for the Simulink.ModelAdvisor object ma.

Input Arguments

ma — Model Advisor object
Simulink.ModelAdvisor object

Simulink.ModelAdvisor object for which you want to obtain the currently selected tasks.

Output Arguments

TaskIDs — Model Advisor Task IDs
cell array of character vectors

Cell array of selected Task IDs for the Simulink.ModelAdvisor object that you specify as an input.

See Also

Simulink.ModelAdvisor
Introduced in R2006a
getTaskAll

Package: Simulink

Get tasks run by Model Advisor

Syntax

TaskIDs = getTaskAll(ma)

Description

Use the TaskIDs = getTaskAll(ma) method to obtain the tasks that are run for the Simulink.ModelAdvisor object ma.

Input Arguments

ma — Model Advisor object
Simulink.ModelAdvisor object

Simulink.ModelAdvisor object for which you want to obtain the tasks that the Model Advisor ran.

Output Arguments

TaskIDs — Model Advisor Task IDs
cell array of character vectors

Cell array of Task IDs that are run for the Simulink.ModelAdvisor object that you specify as an input.

See Also

Simulink.ModelAdvisor
Introduced in R2006a
Simulink.ModelAdvisor.openConfigUI

**Package**: Simulink

Starts Model Advisor Configuration Editor

**Syntax**

Simulink.ModelAdvisor.openConfigUI(ma)

**Description**

Use the `Simulink.ModelAdvisor.openConfigUI(ma)` method to open the Model Advisor Configuration Editor. Use the Model Advisor Configuration Editor to create custom Model Advisor configurations.

The Model Advisor Configuration Editor is available with Simulink Check. For more information, see “Organize Checks and Folders Using the Model Advisor Configuration Editor” (Simulink Check).

Before starting the Model Advisor Configuration Editor, make sure that the current folder is writable. If the folder is not writable, you see an error message when you start the Model Advisor Configuration Editor.

The Model Advisor Configuration Editor uses the `slprj` folder in the “Code generation folder” to store reports and other information. If the `slprj` folder does not exist in the code generation folder, the Model Advisor Configuration Editor creates it. For more information, see “Model Reference Simulation Targets”.

**Input Arguments**

**ma — Model Advisor object**

Simulink.ModelAdvisor object

Simulink.ModelAdvisor object for which you want to open the Model Advisor Configuration Editor.
See Also
Simulink.ModelAdvisor

Introduced in R2006a
Simulink.ModelAdvisor.reportExists

Package: Simulink

Determine whether Model Advisor report exists for model or subsystem

Syntax

exists = Simulink.ModelAdvisor.reportExists(system)

Description

Use the exists = Simulink.ModelAdvisor.reportExists(system) method to determine whether a Model Advisor report exists in the slprj/modeladvisor subfolder of the MATLAB working folder.

Input Arguments

system — Model or subsystem
character vector

Model or subsystem for which you want to determine whether a Model Advisor report exists.

Output Arguments

exists — Boolean value indicating whether Model Advisor report exists
Boolean

Boolean value that indicates whether Model Advisor report exists. A value of 1 indicates that the report exists. A value of 0 indicates that the report does not exist.
See Also
Simulink.ModelAdvisor

Introduced in R2006a
runCheck

Package: Simulink

Run currently selected checks

Syntax

success = runCheck(ma,checkID)

Description

The `success = runCheck(ma,checkID)` method runs selected Model Advisor checks. Invoking this method is equivalent to selecting the **Run Selected Checks** button on the Model Advisor.

Input Arguments

- **ma — Model Advisor object**
  Simulink.ModelAdvisor object

  `Simulink.ModelAdvisor` object for which you want to run Model Advisor.

- **checkID — Model Advisor check IDs**
  character vector | cell array of character vectors

  If you want to run the Model Advisor on a subset of the currently selected Model Advisor checks, specify this optional input parameter.

  Example: `mathworks.design.UnconnectedLinesPorts`
Output Arguments

success — Boolean value indicating whether the Model Advisor ran on selected checks

Boolean

Boolean value that indicates whether the Model Advisor ran on the selected checks. A value of 1 indicates that the Model Advisor successfully ran on the selected checks. A value of 0 indicates that the Model Advisor did not run on the selected checks.

See Also

Simulink.ModelAdvisor

Introduced in R2006a
runTask

Package: Simulink

Run currently selected tasks

Syntax

success = runTask(ma)

Description

The `success = runTask(ma)` method runs selected Model Advisor tasks. Invoking this method is equivalent to the Run Selected Checks button on the Model Advisor.

Input Arguments

`ma` — Model Advisor object
Simulink.ModelAdvisor object

Simulink.ModelAdvisor object for which you want to run the Model Advisor.

Output Arguments

`success` — Boolean value indicating whether the Model Advisor ran selected tasks
Boolean

Boolean value that indicates whether the Model Advisor ran on the selected tasks. A value of 1 indicates that the Model Advisor successfully ran on the selected tasks. A value of 0 indicates that the Model Advisor did not run on the selected tasks.
See Also
Simulink.ModelAdvisor

Introduced in R2006a
selectCheck

Package: Simulink

Select checks

Syntax

success = selectCheck(ma,checkID)

Description

The \(\text{success} = \text{selectCheck}(\text{ma},\text{checkID})\) method selects Model Advisor checks.

Input Arguments

ma — Model Advisor object

Simulink.ModelAdvisor object

Simulink.ModelAdvisor object for which you want to select Model Advisor checks.

checkID — Model Advisor Check IDs

character vector | cell array of character vectors

Model Advisor checks for you to select.

Example: 'mathworks.design.UnconnectedLinesPorts'

Output Arguments

success — Boolean value indicating whether the Model Advisor selected the checks

Boolean
Boolean value that indicates whether the Model Advisor selected the checks. A value of 1 indicates that the Model Advisor successfully selected the specified checks. A value of 0 indicates that the Model Advisor did not select the specified checks.

See Also
Simulink.ModelAdvisor

Introduced in R2006a
selectCheckAll

Package: Simulink
Select all checks

Syntax

success = selectCheckAll(ma)

Description

The success = selectCheckAll(ma) method selects all Model Advisor checks.

Input Arguments

ma — Model Advisor object
Simulink.ModelAdvisor object
Simulink.ModelAdvisor object for which you want to select all Model Advisor checks.

Output Arguments

success — Boolean value indicating whether all Model Advisor checks are selected
Boolean

Boolean value that indicates whether all Model Advisor checks are selected. A value of 1 indicates that the Model Advisor successfully selected all checks. A value of 0 indicates that the Model Advisor did not select all checks.

See Also

Simulink.ModelAdvisor
Introduced in R2006a
selectCheckForGroup

Package: Simulink

Select check group

Syntax

success = selectCheckForGroup(ma,groupName)

Description

The success = selectCheckForGroup(ma,groupName) method selects all Model Advisor checks in the group that you specify.

Input Arguments

ma — Model Advisor object
Simulink.ModelAdvisor object

Simulink.ModelAdvisor object for which you want to select the Model Advisor checks.

groupName — Check group name
character vectorcell array of character vectors

Name of groups that contain the Model Advisor checks that you want to select.

Output Arguments

success — Boolean value indicating whether the Model Advisor selected the checks
Boolean
Boolean value that indicates whether the Model Advisor selected the checks. A value of 1 indicates that the Model Advisor successfully selected the specified checks. A value of 0 indicates that the Model Advisor did not select the specified checks.

See Also
Simulink.ModelAdvisor

Introduced in R2006a
selectCheckForTask

**Package:** Simulink

Select checks for a specific task or tasks

**Syntax**

```matlab
success = selectCheckforTask(ma,taskIDs)
```

**Description**

Use the `success = selectCheckforTask(ma,taskIDs)` method to select the checks for a task or tasks.

**Input Arguments**

- `ma` — Model Advisor object
  *Simulink.ModelAdvisor* object
  
  *Simulink.ModelAdvisor* object for which you want to select checks for a specific task or tasks.

- `taskIDs` — Model Advisor Task IDs
  *cell array of character vectors* | *character vector*
  
  Cell array of Task IDs for the *Simulink.ModelAdvisor* object that you specify as an input.

**Output Arguments**

- `success` — Boolean value indicating whether the checks are selected
  *Boolean*
Boolean value that indicates whether the Model Advisor successfully selected the checks for the specified task or tasks. A value of 1 indicates that the Model Advisor successfully selected the checks. A value of 0 indicates that the Model Advisor did not select the checks.

See Also
Simulink.ModelAdvisor

Introduced in R2006a
selectTask

Package: Simulink

Select Model Advisor task

Syntax

success = selectTask(ma, TaskIDs)

Description

The success = selectTask(ma, TaskIDs) method selects the Model Advisor tasks.

Input Arguments

ma — Model Advisor object
Simulink.ModelAdvisor object

Simulink.ModelAdvisor object for which you want to select Model Advisor tasks.

TaskIDs — Model Advisor Task IDs
cell array of character vectors

Cell array of Task IDs that you want to select for the Simulink.ModelAdvisor object, which you specify as an input.

Output Arguments

success — Boolean value indicating whether Model Advisor selected tasks

Boolean

Boolean value that indicates whether the Model Advisor selected tasks. A value of 1 indicates that the Model Advisor successfully selected the tasks. A value of 0 indicates that the Model Advisor did not select the tasks.
See Also
Simulink.ModelAdvisor

Introduced in R2006a
selectTaskAll

Package: Simulink

Select all Model Advisor tasks

Syntax

success = selectTaskAll(ma)

Description

The success = selectTaskAll(ma) method selects all Model Advisor tasks.

Input Arguments

ma — Model Advisor object
Simulink.ModelAdvisor object

Simulink.ModelAdvisor object for which you want to select the Model Advisor tasks.

Output Arguments

success — Boolean value indicating whether Model Advisor selected tasks

Boolean

Boolean value that indicates whether the Model Advisor selected tasks. A value of 1 indicates that the Model Advisor successfully selected the tasks. A value of 0 indicates that the Model Advisor did not select the tasks.

See Also

Simulink.ModelAdvisor

5-387
Introduced in R2006a
setActionEnable

Package: Simulink
Set status for check action

Syntax

setActionEnable(ma, value)

Description

The `setActionEnable(ma, value)` method enables or disables the Action box. Only a check callback function can invoke this method.

This method enables you to access custom checks that you create by using the Model Advisor customization API, an optional feature that is available with Simulink Check software. For more information, see “Define Custom Checks” (Simulink Check).

Input Arguments

`ma` — Model Advisor object
Simulink.ModelAdvisor object

Simulink.ModelAdvisor object for which you want to set check action status.

`value` — Boolean value indicating whether to enable or disable the Model Advisor Action box

Boolean

Boolean value that indicates whether to enable or disable Model Advisor Action box. To enable the box, specify a value of `true`. To disable the box, specify a value of `false`.

See Also

Simulink.ModelAdvisor
Introduced in R2006a
setBaselineMode

Package: Simulink

Set Model Advisor baseline mode

Syntax

setBaselineMode(ma,mode)

Description

Use the `setBaselineMode(ma,mode)` method to set the Model Advisor baseline mode. Baseline mode causes the Model Advisor verify methods to generate baseline comparison data for verifying the results of a Model Advisor run.

Input Arguments

`ma` — Model Advisor object
`Simulink.ModelAdvisor` object

Simulink.ModelAdvisor object for which you want to set the baseline mode.

`mode` — Boolean value indicating the Model Advisor baseline mode
`Boolean`

Boolean value that indicates the Model Advisor baseline mode. A value of `true` indicates that baseline mode is on. A value of `false` indicates that baseline mode is off.

See Also

`Simulink.ModelAdvisor`

Introduced in R2006a
**setCheckErrorSeverity**

**Package:** Simulink

Set severity of check failure

**Syntax**

`setCheckErrorSeverity(ma,value)`

**Description**

Use the `setCheckErrorSeverity(ma,value)` method to set the check error severity for the currently running check. Only the callback function of a check can invoke this method.

This method enables you to access custom checks that you create by using the Model Advisor customization API, an optional feature that is available with Simulink Check software. For more information, see “Define Custom Checks” (Simulink Check).

**Input Arguments**

- **ma** — Model Advisor object
  
  `Simulink.ModelAdvisor` object
  
  Simulink.ModelAdvisor object for which you want to set the check error severity for the currently running check.

- **value** — Integer indicating severity of failure
  
  `integer`
  
  A value of 0 indicates a warning check result. A value of 1 indicates a failed check result.

**See Also**

`Simulink.ModelAdvisor`
Introduced in R2006a
setCheckResult

Package: Simulink

Set result for currently running check

Syntax

success = setCheckResult(ma,result)

Description

Use the `success = setCheckResult(ma,result)` method to set the check result for the currently running check. Only the callback function of a check can invoke this method.

This method enables you to access custom checks that you create using the Model Advisor customization API, an optional feature that is available with Simulink Check software. For more information, see “Define Custom Checks” (Simulink Check).

Input Arguments

`ma` — Model Advisor object
Simulink.ModelAdvisor object

`ma` — Model Advisor object for which you want to set the result for the currently running check.

`result` — Result to be set
character vector | cell array of character vectors

Character vector or cell array that specifies the result of the currently running check.
Output Arguments

**success** — Boolean value indicating whether the result is set

Boolean value that indicates whether the Model Advisor successfully set the result for the currently running check. A value of 1 indicates that the Model Advisor successfully set the result. A value of 0 indicates that the Model Advisor did not set the result.

See Also

Simulink.ModelAdvisor

Introduced in R2006a
setCheckResultData

Package: Simulink

Set result data for currently running check

Syntax

success = setCheckResultData(ma,data)

Description

Use the success = setCheckResultData(ma,data) method to set the check result data for the currently running check. Only the callback function of a check can invoke this method.

This method enables you to access custom checks that you create with the Model Advisor customization API, an optional feature that is available with Simulink Check software. For more information, see “Define Custom Checks” (Simulink Check).

Input Arguments

ma — Model Advisor object
Simulink.ModelAdvisor object

Simulink.ModelAdvisor object for which you want to set result data for the currently running check.

data — Result data
cell array of character vectors

Result data that you specify for a custom check.
Output Arguments

success — Boolean value indicating whether result data is set

Boolean

Boolean value that indicates whether the Model Advisor successfully set result data for the currently running check. A value of 1 indicates that the Model Advisor successfully set the data. A value of 0 indicates that the Model Advisor did not set the data.

See Also

Simulink.ModelAdvisor

Introduced in R2006a
setCheckResultStatus

Package: Simulink

Set status for currently running check

Syntax

success = setCheckResultStatus(ma,status)

Description

The `success = setCheckResultStatus(ma,status)` method sets the status for the currently running check. Only the callback function of a check can invoke this method.

This method enables you to access custom checks that you create with the Model Advisor customization API, an optional feature that is available with Simulink Check software. For more information, see “Define Custom Checks” (Simulink Check).

Input Arguments

`ma` — Model Advisor object

`Simulink.ModelAdvisor` object

`Simulink.ModelAdvisor` object for which you want to set the status for the currently running check.

`status` — Status of currently running check

`Boolean`

`Boolean` value that indicates the status of the check that just ran. A value of `true` indicates that the check passed. A value of `false` indicates that the check failed.
Output Arguments

success — Boolean value indicating whether the Model Advisor successfully set the status

Boolean

Boolean value that indicates whether the Model Advisor successfully set the status of the currently running check. A value of 1 indicates that the Model Advisor successfully set the status. A value of 0 indicates that the Model Advisor did not set the status.

See Also

Simulink.ModelAdvisor

Introduced in R2006a
**setInputParameters**

**Package:** Simulink

Specify input parameters

**Syntax**

```matlab
success = setInputParameters(ma, checkID, InputParameters)
```

**Description**

Use the `success = setInputParameters(ma, checkID, InputParameters)` method to specify input parameters for a Model Advisor check.

This method enables you to access custom checks that you create using Simulink Check. For more information, see “Define Custom Checks” (Simulink Check).

**Input Arguments**

- **ma** — Model Advisor object
  
  Simulink.ModelAdvisor object
  
  Simulink.ModelAdvisor object for which to specify the input parameters for the specific Model Advisor check.

- **checkID** — Model Advisor check
  
  character vector
  
  Model Advisor check ID to which the input parameters apply. If you use this method inside a current active check function, you can omit this argument.

  Example: `'mathworks.design.UnconnectedLinesPorts'`

- **InputParameters** — Input parameters
  
  cell array of Simulink.ModelAdvisor.InputParameter objects
Cell array that contains the ModelAdvisor.InputParameter objects for a Model Advisor check.

**Output Arguments**

success — Boolean value indicating whether the input parameters were specified for check

Boolean

Boolean value that indicates whether the Model Advisor specified the input parameters. A value of 1 indicates that the Model Advisor successfully specified the input parameters for the check. A value of 0 indicates that the Model Advisor did not specify the input parameters.

**See Also**

ModelAdvisor.InputParameter | Simulink.ModelAdvisor

**Topics**

“Create Model Advisor Checks” (Simulink Check)
“Create and Add Custom Checks - Basic Examples” (Simulink Check)

**Introduced in R2006a**
setListViewParameters

**Package:** Simulink

Specify list-view parameters for check

**Syntax**

```matlab
setListViewParameters(ma,checkID, params)
```

**Description**

The `setListViewParameters(ma,checkID, params)` method sets the list-view parameters for a check. Only the callback function of a check can invoke this method.

This method enables you to access custom checks that you create with the Model Advisor customization API, an optional feature that is available with Simulink Check software. For more information, see “Define Custom Checks” (Simulink Check).

**Input Arguments**

- **ma** — Model Advisor object
  
  `Simulink.ModelAdvisor` object

  Simulink.ModelAdvisor object for which you want to set list-view parameters.

- **checkID** — Model Advisor Check ID
  
  Character vector

  Character vector specifying the identifier for a Model Advisor check.

- **params** — Cell array of ModelAdvisor.ListViewParameter objects
  
  ModelAdvisor.ListViewParameter objects

  Set the list-view parameters for the check.
See Also
ModelAdvisor.ListViewParameter | Simulink.ModelAdvisor

Introduced in R2006a
**verifyCheckRan**

**Package:** Simulink

Verify that Model Advisor ran a set of checks

**Syntax**

```
[success, missingChecks, additionalChecks]= verifyCheckRan(ma, CheckIDs)
```

**Description**

Use the `[success, missingChecks, additionalChecks]= verifyCheckRan(ma, CheckIDs)` method to verify that the Model Advisor ran a set of checks.

**Input Arguments**

- **ma — Model Advisor object**
  Simulink.ModelAdvisor object

  Simulink.ModelAdvisor object for which you want to verify that a set of checks ran

- **CheckIDs — Model Advisor Check IDs**
  cell array of character vectors

  Cell array of character vectors specifying the identifiers of Model Advisor checks that you want to verify have run.

**Output Arguments**

- **success — Boolean value indicating whether the Model Advisor ran checks**
  Boolean
Boolean value that indicates whether the Model Advisor successfully ran a set of checks. A value of 1 indicates that the Model Advisor successfully ran the checks. A value of 0 indicates that the Model Advisor did not run the checks.

**missingChecks — Cell array of Check IDs**  
cell array of character vectors

Cell array of character vectors specifying the identifiers of specified Model Advisor checks that did not run.

**additionalChecks — Model Advisor Check IDs**  
cell array of character vectors

Cell array of character vectors specifying the identifiers of unspecified Model Advisor checks that ran.

**See Also**  
Simulink.ModelAdvisor

**Introduced in R2006a**
verifyCheckResult

**Package:** Simulink

Generate baseline Model Advisor check results file or compare current check results to baseline check results

**Syntax**

```matlab
[success, message] = verifyCheckResult(ma, baseline, CheckIDs)
```

**Description**

Use the `[success, message] = verifyCheckResult(ma, baseline, CheckIDs)` method to generate a baseline Model Advisor check results file or compare to current check results to baseline check results. If the Model Advisor is in baseline mode, this method stores the most recent results of running the checks that you specify by the `CheckIDs` argument in a MAT-file. For the MAT-file, use the location that you specify by using the `baseline` input argument.

To generate a baseline report, put the Model Advisor in baseline mode. Then invoke this method with the baseline argument set to the location where you want to store the baseline results. To perform a current-to-baseline report comparison, first check that the Model Advisor is not in baseline mode. Then, invoke this method with the path of the baseline report as the value of the baseline input argument.

**Input Arguments**

- **ma** — Model Advisor object
  
  Simulink.ModelAdvisor object

  Simulink.ModelAdvisor object for which you want to verify that a set of checks ran.

- **baseline** — Path of the baseline check results MAT-file
  
  character vector
To perform a current-to-baseline report comparison, make sure that the Model Advisor is not in baseline mode and specify a path to the baseline report.

**CheckIDs — Model Advisor Check IDs**
cell array of character vectors

Cell array of character vectors specifying the identifiers of Model Advisor checks that you want to verify by using check results.

**Output Arguments**

**success — Boolean value indicating whether the method succeeded**

Boolean value that indicates whether the method ran successfully. A value of 1 indicates that the method verified that the current check results match the baseline results. A value of 0 indicates that the Model Advisor did not verify check results. If the method is unable to store the check results at the specified location, it returns false in the output variable success. The reason for the failure is in the message argument.

**message — Error message**

content vector

Error message indicating why Simulink did not verify check result.

**See Also**

Simulink.ModelAdvisor

**Introduced in R2006a**
verifyCheckResultStatus

Package: Simulink

Verify that a model passed or failed a set of checks

Syntax

[succes message]= verifyCheckResultStatus(ma, baseline, checkIDs)

Description

Use the [success message]= verifyCheckResultStatus(ma, baseline, checkIDs) method to verify that a model has passed or failed a set of checks.

Before invoking the verifyCheckResultStatus method, run the checks that you specify with checkIDs. Then use the getCheckResultStatus method to obtain the boolean values for the baseline argument.

Input Arguments

ma — Model Advisor object
Simulink.ModelAdvisor

Simulink.ModelAdvisor object for which you want to verify that a set of checks passed.

baseline — Statuses from running most recent round of checks
Cell array of Boolean variables

Statues from the most recent checks run that you specify by Check IDs.

checkIDs — Model Advisor Check IDs
cell array of character vectors
Cell array of character vectors specifying the identifiers of Model Advisor checks that you want to verify have run.

**Output Arguments**

**success** — Boolean value indicating whether the check statuses match the baseline

Boolean value that indicates whether the statuses match the baseline. A value of 1 indicates that the statuses match. A value of 0 indicates that the statuses do not match the baseline.

**message** — Message indicating status

Character vector

Message indicating whether the method successfully verified the check result statuses.

**See Also**

Simulink.ModelAdvisor

**Introduced in R2006a**
verifyHTML

Package: Simulink

Generate baseline Model Advisor report or compare current report to baseline report

Syntax

[succes message] = verifyHTML(ma,baseline)

Description

Use the [success message] = verifyHTML(ma,baseline) method to generate a Model Advisor report or to compare a current report to a baseline report.

This method enables you to compare a Model Advisor report with a baseline report to determine if they differ. You can use the method to generate the baseline report and perform current-to-baseline report comparisons. To generate a baseline report, use the setBaselineMode method to put the Model Advisor in baseline mode. Invoke this method with the baseline argument set to the location where you want to store the baseline report. To perform a current-to-baseline report comparison, first make sure that the Model Advisor is not in baseline mode. Then, invoke this method with the path of the baseline report as the value of the baseline input argument.

Input Arguments

ma — Model Advisor object
Simulink.ModelAdvisor object

Simulink.ModelAdvisor object for which you want to generate a baseline report or compare current report to baseline report

baseline — Path of Model Advisor report
character vector

Path to baseline Model Advisor report
Output Arguments

**success** — Boolean value indicating whether the method is successful

Boolean value that indicates whether the method is successful. If the Model Advisor is in baseline mode, this method stores the most recently generated Model Advisor report at the location specified by the `baseline` input argument. If the method is unable to store a copy of the report at the specified location, it returns `false` in the output variable `success`. The reason for the failure is in the output variable `message`. If the Model Advisor is not in baseline mode, this method compares the report most recently generated by the Model Advisor with the report specified by `baseline`. If the current report has the same content as the baseline report, this method returns `true` as the value of the `success` output variable.

**message** — Message indicating status

Character vector

Message indicating whether the Model Advisor is able to successfully store a report.

See Also

`Simulink.ModelAdvisor`

Introduced in R2006a
Simulink.ModelDataLogs

Container for signal data logs of a model

Description

Note The ModelDataLogs class is supported for backwards compatibility. Starting in R2016a, you cannot log data in the ModelDataLogs format. Signal logging uses the Dataset format. In R2016a or later, when you simulate a model from an earlier release that had used ModelDataLogs format, the model uses Dataset format to log data.

You can convert signal logging data from ModelDataLogs to Dataset format. Converting to Dataset format can facilitate post-processing with other logged data (for example, logged states), which can also use Dataset format. For more information, see “Convert Logged Data to Dataset Format”.

If you have legacy code that uses ModelDataLogs, you can encounter situations that require updates to your code or model. For more information about updating scripts, see “Migrate Scripts That Use Legacy ModelDataLogs API”.

In releases before R2016a, when you set Configuration Parameters > Data Import/Export > Signal logging format to ModelDataLogs, logging data produces one or more Simulink.ModelDataLogs objects that contain logged data. Logging creates an instance of this class for the top model and for each model referenced by the top model that contains logged signals. The ModelDataLogs object for the top model is assigned to a variable in the base workspace. The name of the variable is specified by the Configuration Parameters > Data Import/export > Signal logging name parameter. The default value is logsout.

A ModelDataLogs object has a variable number of properties. The first property, Name, specifies the name of the model whose signal data the object contains or, if the model is a referenced model, the name of the Model block that references the model. The remaining properties reference objects that contain signal data logged during simulation of the model. The objects may be instances of these types of objects:

- Simulink.ModelDataLogs
Container for the data logs of a model

- **Simulink.SubsysDataLogs**

  Container for the data logs of a subsystem

- **Simulink.Timeseries**

  Data log for any signal except a mux or bus signal

- **Simulink.TsArray**

  Data log for a mux or bus signal

The names of the properties identify the data being logged as follows:

- For signal data logs, the name of the signal
- For a subsystem or model log container, the name of the subsystem or model, respectively

Consider, for example, the following model.
As indicated by the testpoint icons, this model specifies that Simulink software should log the signals named step and scope in the root system and the signal named clk in the subsystem named Delayed Out. After you simulate this model in a release earlier than R2016a, the MATLAB workspace contains the following variable:

```matlab
Simulink.ModelDataLogs (siglgex):
<table>
<thead>
<tr>
<th>Name</th>
<th>elements</th>
<th>Simulink Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>scope</td>
<td>2</td>
<td>TsArray</td>
</tr>
<tr>
<td>step</td>
<td>1</td>
<td>Timeseries</td>
</tr>
<tr>
<td>('Delayed Out')</td>
<td>2</td>
<td>SubsysDataLogs</td>
</tr>
</tbody>
</table>
```

You can use fully qualified object names or the Simulink unpack command to access the signal data. For example, to access the amplitudes of the clk signal in the Delayed Out subsystem in a logsout object, enter

```matlab
data = logsout.('Delayed Out').clk.Data;
```

or

```matlab
>> logsout.unpack('all');
>> data = clk.Data;
```

## Access Logged Signal Data Saved in ModelDataLogs Format

The `Simulink.ModelDataLogs` object contains signal data objects to capture signal logging information for specific model elements.

<table>
<thead>
<tr>
<th>Model Element</th>
<th>Signal Data Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>Top-level or referenced model</td>
<td><code>Simulink.ModelDataLogs</code></td>
</tr>
<tr>
<td>Subsystem in a model</td>
<td><code>Simulink.SubsysDataLogs</code></td>
</tr>
<tr>
<td>Signal other than a bus or Mux signal</td>
<td><code>Simulink.Timeseries</code></td>
</tr>
<tr>
<td>Bus signal or Mux signal</td>
<td><code>Simulink.TsArray</code></td>
</tr>
</tbody>
</table>

## Handling Spaces and Newlines in Logged Names

Signal names in data logs can have spaces or newlines in their names when the signal:

- Is named and the name includes a space or newline character.
• Is unnamed and originates in a block whose name includes a space or newline character.
• Exists in a subsystem or referenced model, and the name of the subsystem, Model block, or of any superior block includes a space or newline character.

The following model shows a signal whose name contains a space, a signal whose name contains a newline, and an unnamed signal that originates in a block whose name contains a newline:

![Model Diagram](image)

The following example shows how to handle spaces or new lines in logged names, if a model uses `Simulink.ModelDataLogs` for the signal logging format.

```matlab
logsnout = Simulink.ModelDataLogs (model_name);
Name     Elements  Simulink Class
('x y')                   1      Timeseries
('a
b')                   1      Timeseries
('SL_Sine Wave1')         1      Timeseries
```

You cannot access any of the `Simulink.Timeseries` objects in this log using TAB name completion or by typing the name to MATLAB. This syntax is not recognized because the
space or newline in each name appears to the MATLAB parser as a separator between identifiers. For example:

```
logsout.x y
```

```matlab
??? logsout.x y
| Error: Unexpected MATLAB expression.
```

To reference a `Simulink.Timeseries` object whose name contains a space, enclose the element containing the space in single quotes:

```
logsout.('x y')
```

To reference a `Simulink.Timeseries` object whose name contains a newline, concatenate to construct the element containing the newline:

```
cr=sprintf('
')
logsout.([...'a' cr 'b'])
```

The same techniques work when a space or newline in a data log derives from the name of:

- An unnamed logged signal's originating block
- A subsystem or Model block that contains any logged signal
- Any block that is superior to such a block in the model hierarchy

This code can reference logged data for the signal:

```
logsout.([...'SL_Sine' cr 'Wave1'])
```

For names with multiple spaces, newlines, or both, repeat and combine the two techniques as needed to specify the intended name to MATLAB.
Bus Signals

ModelDataLogs format stores each logged bus signal data in a separate Simulink.TsArray object.

The hierarchy of a bus signal is preserved in the logged signal data. The logged name of a signal in a virtual bus derives from the name of the source signal. The logged name of a signal in a nonvirtual bus derives from the applicable bus object, and can differ from the name of the source signal. See “Composite Signals” for information about those capabilities.

See Also

“Convert Logged Data to Dataset Format”, “Migrate Scripts That Use Legacy ModelDataLogs API”, Simulink.SubsysDataLogs, Simulink.Timeseries, Simulink.TsArray, who, whos, unpack

Introduced before R2006a
Simulink.SimState.ModelSimState class

Package: Simulink.SimState

(Not recommended) Access SimState snapshot data

Note Simulink.SimState.ModelSimState is not recommended. Use Simulink.op.ModelOperatingPoint instead.

Description

The Simulink.SimState.ModelSimState class contains all of the information associated with a “snapshot” of a simulation, including the logged states, the time of the snapshot, and the start time of the simulation. To access these data for a block, use the getBlockSimState method or the loggedStates property.

Properties

description

Specify a description. By default, Simulink generates a character vector based on your model name.

loggedStates

The logged states are the continuous and discrete states of the blocks in a model. These states represent a subset of the complete simulation state (SimState) of the model.

If loggedStates is in Dataset format, you cannot assign a structure or a Simulink.SimulationData.Dataset object with a different number of elements than that of the Dataset object used for loggedStates.

If the loggedStates is in Structure format, you cannot assign a Dataset object.

Attributes:
loggedStates is obtained from the saved states of the block. loggedStates depends on the full state being saved in the SimState object, unlike, properties like description, which are independent of the save states.

**snapshotTime**

Time at which Simulink takes a “snapshot” of the complete simulation states. This data is read only.

**startTime**

Time at which the simulation starts. This data is read only.

**Methods**

getBlockSimState (Not recommended) Access SimState of individual Stateflow Chart, MATLAB Function, or S-function block

setBlockSimState (Not recommended) Set SimState of individual Stateflow Chart, MATLAB Function, or S-function block

**Copy Semantics**

Value. To learn how value classes affect copy operations, see Copying Objects (MATLAB) in the MATLAB Programming Fundamentals documentation.
Simulink.ModelWorkspace

Interact with the model workspace of a model programmatically

Description

Use a Simulink.ModelWorkspace object to interact with a model workspace. For example, you can add and remove variables, set the data source of the workspace, and save changes to the workspace.

For more information, see “Model Workspaces”.

Creation

To create a Simulink.ModelWorkspace, use the get_param function to query the value of the model parameter ModelWorkspace. For example, to create an object named mdlWks that represents the model workspace of a model named myModel.slx:

mdlWks = get_param('myModel','ModelWorkspace')

Properties

DataSource — Source for initializing variables in model workspace

'Model File' (default) | 'MAT-File' | 'MATLAB Code' | 'MATLAB File'

Source for initializing the variables in the model workspace, specified as one of these character vectors:

- 'Model File' — The variables are stored in the model file. When you save the model, you also save the variables.
- 'MATLAB Code' — The variables are created by MATLAB code that you write and store in the model file.
- 'MAT-File' — The variables are stored in a MAT-file, which you can manage and manipulate separately from the model file.
• 'MATLAB File' — The variables are created by MATLAB code in a script file, which you can manage and manipulate separately from the model file.

Data Types: char

**FileName** — Name of external file that stores or creates variables

' ' (empty character vector) (default) | character vector

Name of the external file that stores or creates variables, specified as a character vector. To enable this property, set **DataSource** to 'MAT-File' or 'MATLAB File'.

Example: 'myFile.mat'
Example: 'myFile.m'

Data Types: char

**MATLABCode** — MATLAB code for initializing variables

' ' (empty character vector) (default) | character vector

MATLAB code for initializing variables, specified as a character vector. To enable this property, set **DataSource** to 'MATLAB Code'.

Example: sprintf('%% Create variables that this model uses.

K = 0.00983;
P = Simulink.Parameter(5);')

Data Types: char

**Object Functions**

- `getVariable` — Return value of variable in the model workspace of a model
- `getVariablePart` — Get value of variable property in model workspace
- `setVariablePart` — Set property of variable in model workspace
- `hasVariable` — Determine whether variable exists in the model workspace of a model
- `whos` — Return list of variables in the model workspace of a model
- `saveToSource` — Save model workspace changes to the external data source of the model workspace
- `save` — Save contents of model workspace to a MAT-file
- `reload` — Reinitialize variables from the data source of a model workspace
- `evalin` — Evaluate expression in the model workspace of a model
- `clear` — Clear variables from the model workspace of a model
- `assignin` — Assign value to variable in the model workspace of a model
Examples

Interact With Model Workspace Programmatically

Create a variable in the model workspace of a model. Then, modify the variable and query the variable value to confirm the modification.

Open the example model vdp.

```
open_system('vdp')
```

Create a `Simulink.ModelWorkspace` object `mdlWks` that represents the model workspace of `vdp`.

```
mdlWks = get_param('vdp','ModelWorkspace');
```

Create a variable named `myVar` with value `5.12` in the model workspace.

```
assignin(mdlWks,'myVar',5.12)
```

Apply a new value, `7.22`. To do so, first create a temporary copy of the variable by using the `getVariable` function. Then, modify the copy and use it to overwrite the original variable in the model workspace.

```
temp = getVariable(mdlWks,'myVar');
temp = 7.22;
assignin(mdlWks,'myVar',temp)
```

Confirm the new value by querying the value of the variable.

```
getVariable(mdlWks,'myVar')
```

```
ans =

7.2200
```

See Also

Topics

“Model Workspaces”
“Manage Design Data”

Introduced before R2006a
assignin

**Package:** Simulink

Assign value to variable in the model workspace of a model

**Syntax**

assignin(mdlWks,varName,varValue)

**Description**

assignin(mdlWks,varName,varValue) assigns the value varValue to the MATLAB variable varName in the model workspace represented by the Simulink.ModelWorkspace object mdlWks. If the variable does not exist, assignin creates it.

**Examples**

**Assign Value to Variable in Model Workspace**

Open the example model vdp.

```matlab
open_system('vdp')
```

Create a Simulink.ModelWorkspace object that represents the model workspace of vdp.

```matlab
mdlWks = get_param('vdp','ModelWorkspace');
```

Create a variable named myVar with value 5.12 in the model workspace.
assignin(mdlWks,'myVar',5.12)

**Input Arguments**

**mdlWks** — Target model workspace
Simulink.ModelWorkspace object

Target model workspace, specified as a Simulink.ModelWorkspace object.

**varName** — Name of target variable
character vector

Name of the target variable, specified as a character vector.

Example: 'myVar'
Data Types: char

**varValue** — Value to assign to target variable
valid value

Value to assign to the target value, specified as a valid value. For example, you can specify a literal number, a structure, or an expression that evaluates to a valid value.

If you specify the name of a handle object, such as a Simulink.Parameter object, use the copy function to create a separate copy of the object.

Example: 5.12
Example: struct('a',5.12,'b',7.22)
Example: Simulink.Parameter(5.12)
Example: copy(myExistingParameterObject)

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | logical | char | string | struct | table | cell | categorical | datetime | duration | calendarDuration | fi
Complex Number Support: Yes

**See Also**

Simulink.ModelWorkspace
Introduced before R2006a
clear

Package: Simulink

Clear variables from the model workspace of a model

Syntax

clear(mdlWks)

Description

clear(mdlWks) removes all variables from the model workspace represented by the Simulink.ModelWorkspace object mdlWks.

Examples

Clear Variables From Model Workspace

Open the example model vdp.

open_system('vdp')

Create a Simulink.ModelWorkspace object that represents the model workspace of vdp.

mdlWks = get_param('vdp','ModelWorkspace');

Create a variable named myVar with value 5.12 in the model workspace.

assignin(mdlWks,'myVar',5.12)

Clear all variables from the model workspace, including myVar.
clear(mdlWks)

**Input Arguments**

*mdlWks* — Target model workspace
Simulink.ModelWorkspace object

Target model workspace, specified as a Simulink.ModelWorkspace object.

**See Also**

Simulink.ModelWorkspace

*Introduced before R2006a*
**evalin**

**Package: Simulink**

Evaluate expression in the model workspace of a model

**Syntax**

```matlab
result = evalin(mdlWks,expression)
```

**Description**

`result = evalin(mdlWks,expression)` evaluates the expression `expression` in the model workspace represented by the `Simulink.ModelWorkspace` object represented by `mdlWks`. The function returns the result of the expression in `result`.

**Note** For setting and getting variable properties in the model workspace, consider using `setVariablePart` and `getVariablePart` instead of `evalin` because:

- The functions do not create new variables or cause unintended results.
- `getVariablePart` does not dirty the model.

For information on these alternatives, see `Simulink.ModelWorkspace`.

**Examples**

**Evaluate Expression in Model Workspace**

Open the example model `vdp`.

```matlab
open_system('vdp')
```

Create a `Simulink.ModelWorkspace` object that represents the model workspace of `vdp`. 
mdlWks = get_param('vdp','ModelWorkspace');

Create some variables in the model workspace.

assignin(mdlWks,'myVar',5.12)
assignin(mdlWks,'myOtherVar',7.22)

Evaluate the expression myLastVar = myVar + myOtherVar in the model workspace. The expression creates another variable, myLastVar, whose value is the sum of the first two variables.

evalin(mdlWks,'myLastVar = myVar + myOtherVar');

**Input Arguments**

- **mdlWks** — Target model workspace
  Simulink.ModelWorkspace object
  Target model workspace, specified as a Simulink.ModelWorkspace object.

- **expression** — Expression to evaluate
  character vector
  Expression to evaluate, specified as a character vector.
  Example: 'myLastVar = myVar + myOtherVar'
  Data Types: char

**Output Arguments**

- **result** — Result of expression
  number, structure, or other MATLAB value
  Result of the evaluated expression, returned as a number, structure, or other MATLAB value.

**See Also**
Simulink.ModelWorkspace
Introduced before R2006a
reload

Package: Simulink

Reinitialize variables from the data source of a model workspace

Syntax

reload(mdlWks)

Description

reload(mdlWks) reinitializes the variables in the model workspace represented by the Simulink.ModelWorkspace object mdlWks. When you set the DataSource property of the model workspace to 'MAT-File', 'MATLAB File', or 'MATLAB Code', reload overwrites variables that exist in the model workspace by loading the associated MAT-file or by running the associated MATLAB code.

Examples

Reinitialize Variables in a Model Workspace

Open the example model vdp.

open_system('vdp')

Create a Simulink.ModelWorkspace object that represents the model workspace of vdp.

mdlWks = get_param('vdp','ModelWorkspace');

Configure the model workspace to use some MATLAB code as a data source.

mdlWks.DataSource = 'MATLAB Code';
mdlWks.MATLABCode = sprintf('myVar = 5.12;\nmyOtherVar = 7.22;');
Create variables in the model workspace by executing the MATLAB code.

```matlab
reload(mdlWks)
```

Assign new values to the variables in the model workspace.

```matlab
assignin(mdlWks,'myVar',5.22)
assignin(mdlWks,'myOtherVar',7.33)
```

Overwrite the new values with the values specified by the MATLAB code.

```matlab
reload(mdlWks)
```

Confirm that the variables have the values specified by the MATLAB code.

```matlab
myVarValue = getVariable(mdlWks,'myVar')
myOtherVarValue = getVariable(mdlWks,'myOtherVar')
```

```matlab
myVarValue =
    5.1200

myOtherVarValue =
    7.2200
```

**Input Arguments**

- `mdlWks` — Target model workspace
  - Simulink.ModelWorkspace object

  Target model workspace, specified as a Simulink.ModelWorkspace object.

**See Also**

- Simulink.ModelWorkspace

**Introduced before R2006a**
save

Package: Simulink

Save contents of model workspace to a MAT-file

Syntax

save(mdlWks,fileName)

Description

save(mdlWks,fileName) saves the variables in the model workspace represented by the Simulink.ModelWorkspace object mdlWks to the MAT-file specified by fileName.

When you set the DataSource property of the model workspace to 'MAT-File' or 'MATLAB File', to save to the file that acts as the external data source of the model, use saveToSource instead of save.

Examples

Save Contents of Model Workspace to MAT-File

Open the example model vdp.

dlmWks = get_param('vdp','ModelWorkspace');

Create some variables in the model workspace.

assignin(mdlWks,'myVar',5.12)
assignin(mdlWks,'myOtherVar',7.22)
Save the variables to a new MAT-file named `myVars.mat`.

```matlab
save(mdlWks,'myVars.mat')
```

The MAT-file appears in your current folder.

**Input Arguments**

- **mdlWks** — Target model workspace
  ```matlab```
  Simulink.ModelWorkspace object
  ```
  Target model workspace, specified as a Simulink.ModelWorkspace object.

- **fileName** — Name of target MAT-file
  ```matlab```
  character vector
  ```
  Name of the target MAT-file, specified as a character vector.
  Example: 'myFile.mat'
  Data Types: char

**See Also**

- `Simulink.ModelWorkspace`

**Introduced before R2006a**
saveToSource

Package: Simulink

Save model workspace changes to the external data source of the model workspace

Syntax

saveToSource(mdlWks)

Description

saveToSource(mdlWks) saves the variables in the model workspace represented by the Simulink.ModelWorkspace object mdlWks to the MAT-file or script file specified by the FileName property of the model workspace.

When you set the DataSource property of the model workspace to 'MAT-File' or 'MATLAB File', the FileName property specifies the name of the file that acts as the external data source of the workspace. As you make changes to the variables in the model workspace, use saveToSource to permanently save the changes in the external data source.

Examples

Save Variables to External Data Source of Model Workspace

Open the example model vdp.

open_system('vdp')

Create a Simulink.ModelWorkspace object that represents the model workspace of vdp.

mdlWks = get_param('vdp','ModelWorkspace');

Create some variables in the model workspace.
assignin(mdlWks,'myVar',5.12)
assignin(mdlWks,'myOtherVar',7.22)

Configure the model workspace to use a MAT-file named `myVars.mat` as the data source.

mdlWks.DataSource = 'MAT-File';
mdlWks.FileName = 'myVars.mat';

Save the variables to the external data source (the MAT-file).

saveToSource(mdlWks)

The file appears in your current folder.

**Input Arguments**

- `mdlWks` — Target model workspace
  
  Simulink.ModelWorkspace object

  Target model workspace, specified as a Simulink.ModelWorkspace object.

**See Also**

- [Simulink.ModelWorkspace](#)

**Introduced before R2006a**
**whos**

**Package:** Simulink

Return list of variables in the model workspace of a model

**Syntax**

\[ \text{varList} = \text{whos}(\text{mdlWks}) \]

**Description**

\[ \text{varList} = \text{whos}(\text{mdlWks}) \] returns a list of the variables in the model workspace represented by the `Simulink.ModelWorkspace` object `mdlWks`.

**Examples**

**Return List of Variables in Model Workspace**

Open the example model `vdp`.

```matlab
open_system('vdp')
```

Create a `Simulink.ModelWorkspace` object that represents the model workspace of `vdp`.

```matlab
mdlWks = get_param('vdp','ModelWorkspace');
```

Create some variables in the model workspace.

```matlab
assignin(mdlWks,'myVar',5.12)
assignin(mdlWks,'myOtherVar',7.22)
```

Display a list of the variables in the model workspace.

```matlab
whos(mdlWks)
```
<table>
<thead>
<tr>
<th>Name</th>
<th>Size</th>
<th>Bytes</th>
<th>Class</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>myOtherVar</td>
<td>1x1</td>
<td>8</td>
<td>double</td>
<td></td>
</tr>
<tr>
<td>myVar</td>
<td>1x1</td>
<td>8</td>
<td>double</td>
<td></td>
</tr>
</tbody>
</table>

**Input Arguments**

**mdlWks** — Target model workspace
Simulink.ModelWorkspace object

Target model workspace, specified as a Simulink.ModelWorkspace object.

**Output Arguments**

**varList** — List of variables
nested structure array

List of variables, returned as a nested structure array. For details about the information in the list, see `whos`.

**See Also**

Simulink.ModelWorkspace

`Introduced before R2006a`
getVariable

**Package:** Simulink

Return value of variable in the model workspace of a model

**Syntax**

```matlab
varValue = getVariable(mdlWks,varName)
```

**Description**

`varValue = getVariable(mdlWks,varName)` returns the value of the variable whose name is `varName` that exists in the model workspace represented by the `Simulink.ModelWorkspace` object `mdlWks`.

If the value of the target variable is a handle to a handle object (such as `Simulink.Parameter`), `getVariable` returns a copy of the handle. Changes you make to the variable in the model workspace or to the returned variable (`variableValue`) affect both variables.

To return a deep copy of the handle object, use the `copy` method of the object. To modify a handle object that you store in a model workspace, it is a best practice to use both the `getVariable` and `assignin` methods (see “Modify Property Value of Handle Object” on page 5-441).

**Examples**

**Return Value of Variable in Model Workspace**

Open the example model `vdp`.

```matlab
open_system('vdp')
```
Create a Simulink.ModelWorkspace object that represents the model workspace of vdp.

```matlab
mdlWks = get_param('vdp','ModelWorkspace');
```

Create a variable named myVar in the model workspace.

```matlab
assignin(mdlWks,'myVar',5.12)
```

Return the value of the new variable. Store the value in another variable named varValue.

```matlab
varValue = getVariable(mdlWks,'myVar');
```

**Modify Property Value of Handle Object**

Modify a property of the Simulink.Parameter object K, which is defined in model mdl.slx. When you call getVariable, use the copy method because Simulink.Parameter is a handle class.

```matlab
wksp = get_param(mdl,'ModelWorkspace');
value = copy(getVariable(wksp,'K'));
value.DataType = 'single';
assignin(wksp,'K',value);
```

**Input Arguments**

- **mdlWks** — Target model workspace
  Simulink.ModelWorkspace object
  Target model workspace, specified as a Simulink.ModelWorkspace object.

- **varName** — Name of target variable
  character vector
  Name of the target variable, specified as a character vector.
  Example: 'myVariable'
  Data Types: char
Output Arguments

`varValue — Value of target variable`  
number, structure, or other MATLAB value

Value of the target variable, returned as a number, structure, or other MATLAB value.

See Also

Simulink.ModelWorkspace | get_param

Introduced in R2012a
getVariablePart

Package: Simulink

Get value of variable property in model workspace

Syntax

varValue = getVariablePart(mdlWks,varName.Property)

Description

varValue = getVariablePart(mdlWks,varName.Property) returns the value of the variable property named varName.Property that exists in the model workspace represented by the Simulink.ModelWorkspace object mdlWks.

If the value of the variable property is a handle to a handle object (such as Simulink.Parameter), getVariablePart returns a copy of the handle.

Using getVariablePart is preferable to using evalin for getting variable properties in the model workspace because:

- The function does not dirty the model.
- Use of the function does not result in the creation of a new variable or other unintended results.

Examples

Return Value of Variable Properties in Model Workspace

Open the example model vdp.

open_system('vdp')
Create a `Simulink.ModelWorkspace` object that represents the model workspace of `vdp`.

```matlab
mdlWks = get_param('vdp','ModelWorkspace');
```

Create a structure named `myStruct` with fields `a`, `b`, and `c`.

```matlab
aStruct.a = 10;
aStruct.b = {1,2,3,4,5};
aStruct.c = Simulink.Parameter(7);
mdlWks.assignin('myStruct',aStruct);
```

Return the values of the structure fields. Store the values in `varValuea`, `varValueb`, and `varValuec`.

```matlab
varValuea = getVariablePart(mdlWks,'myStruct.a');
varValueb = getVariablePart(mdlWks,'myStruct.b{1}');
varValueC = getVariablePart(mdlWks,'myStruct.c.Value');
```

## Input Arguments

- **mdlWks** — Target model workspace
  - Type: `Simulink.ModelWorkspace` object
  - Description: Target model workspace, specified as a `Simulink.ModelWorkspace` object.

- **varName.Property** — Name of target variable property
  - Type: `char` character vector
  - Description: Name of the target variable property, specified as a character vector.
  - Example: `'myVariable.Property'`
  - Data Types: `char`

## Output Arguments

- **varValue** — Value of variable property
  - Type: Number, structure, or other MATLAB value
  - Description: Value of the variable property, returned as a number, structure, or other MATLAB value.
If the value of the variable property is a handle to a handle object (such as Simulink.Parameter), `getVariablePart` returns a copy of the handle.

**See Also**
Simulink.ModelWorkspace

**Introduced in R2018b**
setVariablePart

**Package:** Simulink

Set property of variable in model workspace

**Syntax**

```
varValue = setVariablePart(mdlWks, varName.Property, varValue)
```

**Description**

`varValue = setVariablePart(mdlWks, varName.Property, varValue)` assigns `varValue` to the MATLAB variable property `varName.Property` in the model workspace represented by the `Simulink.ModelWorkspace` object `mdlWks`.

Using `setVariablePart` is preferable to using `evalin` for assigning variable properties in the model workspace because the `setVariablePart` function does not create a new variable or cause unintended results.

**Examples**

**Assign Value to Variable Properties in Model Workspace**

Open the example model `vdp`.

```
open_system('vdp')
```

Create a `Simulink.ModelWorkspace` object that represents the model workspace of `vdp`.

```
mdlWks = get_param('vdp','ModelWorkspace');
```

Create a structure named `myStruct` with fields `a`, `b`, and `c`. 
aStruct.a = 10;
aStruct.b = {1,2,3,4,5};
aStruct.c = Simulink.Parameter(7);
mdlWks.assignin('myStruct',aStruct);

Assign new values to the structure fields.

setVariablePart(mdlWks,'myStruct.a', 2);
setVariablePart(mdlWks,'myStruct.b{1}', 2);
setVariablePart(mdlWks,'myStruct.c', Simulink.Parameter(2));

Return the new values of the structure fields. Store the values in varValuea, varValueb, and varValuec.

varValuea = getVariablePart(mdlWks,'myStruct.a')
varValueb = getVariablePart(mdlWks,'myStruct.b{1}')
varValueC = getVariablePart(mdlWks,'myStruct.c.Value')

### Input Arguments

**mdlWks — Target model workspace**  
Simulink.ModelWorkspace object

Target model workspace, specified as a Simulink.ModelWorkspace object.

**varName.Property — Name of target variable property**  
character vector

Name of the target variable property, specified as a character vector.

Example: 'myVariable.Property'

Data Types: char | string

**varValue — Value to assign to variable property**  
valid value

Value to assign to the value property, specified as a valid value. For example, you can specify a literal number, a structure, or an expression that evaluates to a valid value.

If you specify the name of a handle object, such as a Simulink.Parameter object, use the copy function to create a separate copy of the object.
Example: 5.12
Example: struct('a',5.12,'b',7.22)
Example: Simulink.Parameter(5.12)
Example: copy(myExistingParameterObject)

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 |
| uint32 | uint64 | logical | char | string | struct | table | cell | categorical |
| datetime | duration | calendarDuration | fi
Complex Number Support: Yes

See Also
Simulink.ModelWorkspace

Introduced in R2018b
hasVariable

Package: Simulink

Determine whether variable exists in the model workspace of a model

Syntax

varExists = hasVariable(mdlWks,varName)

Description

varExists = hasVariable(mdlWks,varName) returns 1 if a variable whose name is varName exists in the model workspace represented by the Simulink.ModelWorkspace object mdlWks.

Examples

Determine Existence of Variable in Model Workspace

Open the example model vdp.

open_system('vdp')

Create a Simulink.ModelWorkspace object that represents the model workspace of vdp.

mdlWks = get_param('vdp','ModelWorkspace');

Create a variable named myVar in the model workspace.

assignin(mdlWks,'myVar',5.12)

Determine whether a variable named myVar exists in the model workspace.

exists = hasVariable(mdlWks,'myVar')
exists = 1

Input Arguments

`mdlWks` — Target model workspace
Simulink.ModelWorkspace object

Target model workspace, specified as a `Simulink.ModelWorkspace` object.

`varName` — Name of target variable
character vector

Name of the target variable, specified as a character vector.
Example: 'myVariable'
Data Types: char

Output Arguments

`varExists` — Indication of existence
1 | 0

Indication of variable existence, returned as 1 (true) or 0.

See Also
Simulink.ModelWorkspace | get_param

Introduced in R2012a
Simulink.MSFcnRunTimeBlock

Get run-time information about Level-2 MATLAB S-function block

Description

This class allows a Level-2 MATLAB S-function or other MATLAB program to obtain information from Simulink software and provide information to Simulink software about a Level-2 MATLAB S-Function block. Simulink software creates an instance of this class for each Level-2 MATLAB S-Function block in a model. Simulink software passes the object to the callback methods of Level-2 MATLAB S-functions when it updates or simulates a model, allowing the callback methods to get and provide block-related information to Simulink software. See “Write Level-2 MATLAB S-Functions” for more information.

You can also use instances of this class in MATLAB programs to obtain information about Level-2 MATLAB S-Function blocks during a simulation. See “Access Block Data During Simulation” for more information.

The Level-2 MATLAB S-function template `matlabroot/toolbox/simulink/blocks/msfuntmpl.m` shows how to use a number of the following methods.

Parent Class

Simulink.RunTimeBlock

Derived Classes

None
## Property Summary

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>“AllowSignalsWithMoreThan2D” on page 5-454</td>
<td>enable Level-2 MATLAB S-function to use multidimensional signals.</td>
</tr>
<tr>
<td>“DialogPrmsTunable” on page 5-454</td>
<td>Specifies which of the S-function's dialog parameters are tunable.</td>
</tr>
<tr>
<td>“NextTimeHit” on page 5-455</td>
<td>Time of the next sample hit for variable sample time S-functions.</td>
</tr>
</tbody>
</table>

## Method Summary

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>“AutoRegRuntimePrms” on page 5-455</td>
<td>Register this block's dialog parameters as run-time parameters.</td>
</tr>
<tr>
<td>“AutoUpdateRuntimePrms” on page 5-455</td>
<td>Update this block's run-time parameters.</td>
</tr>
<tr>
<td>“IsDoingConstantOutput” on page 5-456</td>
<td>Determine whether the current simulation stage is the constant sample time stage.</td>
</tr>
<tr>
<td>“IsMajorTimeStep” on page 5-457</td>
<td>Determine whether the current simulation time step is a major time step.</td>
</tr>
<tr>
<td>“IsSampleHit” on page 5-457</td>
<td>Determine whether the current simulation time is one at which a task handled by this block is active.</td>
</tr>
<tr>
<td>“IsSpecialSampleHit” on page 5-458</td>
<td>Determine whether the current simulation time is one at which multiple tasks handled by this block are active.</td>
</tr>
<tr>
<td>“RegBlockMethod” on page 5-458</td>
<td>Register a callback method for this block.</td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------------------------------------------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>“RegisterDataTypeFxpBinaryPoint” on page 5-459</td>
<td>Register fixed-point data type with binary point-only scaling.</td>
</tr>
<tr>
<td>“RegisterDataTypeFxpFSlopeFixexpBias” on page 5-460</td>
<td>Register fixed-point data type with [Slope Bias] scaling specified in terms of fractional slope, fixed exponent, and bias.</td>
</tr>
<tr>
<td>“RegisterDataTypeFxpSlopeBias” on page 5-462</td>
<td>Register data type with [Slope Bias] scaling.</td>
</tr>
<tr>
<td>“SetAccelRunOnTLC” on page 5-463</td>
<td>Specify whether to use this block's TLC file to generate the simulation target for the model that uses it.</td>
</tr>
<tr>
<td>“SetPreCompInpPortInfoToDynamic” on page 5-464</td>
<td>Set precompiled attributes of this block's input ports to be inherited.</td>
</tr>
<tr>
<td>“SetPreCompOutPortInfoToDynamic” on page 5-464</td>
<td>Set precompiled attributes of this block's output ports to be inherited.</td>
</tr>
<tr>
<td>“SetPreCompPortInfoToDefaults” on page 5-464</td>
<td>Set precompiled attributes of this block's ports to the default values.</td>
</tr>
<tr>
<td>“SetSimViewingDevice” on page 5-465</td>
<td>Specify whether block is a viewer.</td>
</tr>
<tr>
<td>“SupportsMultipleExecInstances” on page 5-465</td>
<td></td>
</tr>
<tr>
<td>“WriteRTWParam” on page 5-466</td>
<td>Write custom parameter information to Simulink Coder file.</td>
</tr>
</tbody>
</table>
Properties

AllowSignalsWithMoreThan2D

Description
Allow Level-2 MATLAB S-functions to use multidimensional signals. You must set the AllowSignalsWithMoreThan2D property in the setup method.

Data Type
Boolean

Access
RW

DialogPrmsTunable

Description
Specifies whether a dialog parameter of the S-function is tunable. Tunable parameters are registered as run-time parameters when you call the “AutoRegRuntimePrms” on page 5-455 method. Note that SimOnlyTunable parameters are not registered as run-time parameters. For example, the following lines initializes three dialog parameters where the first is tunable, the second in not tunable, and the third is tunable only during simulation.

```matlab
block.NumDialogPrms = 3;
block.DialogPrmsTunable = {'Tunable','Nontunable','SimOnlyTunable'};
```

Data Type
array

Access
RW
**NextTimeHit**

**Description**
Time of the next sample hit for variable sample-time S-functions.

**Data Type**
double

**Access**
RW

**Methods**

**AutoRegRuntimePrms**

**Purpose**
Register a block's tunable dialog parameters as run-time parameters.

**Syntax**
AutoRegRuntimePrms;

**Description**
Use in the PostPropagationSetup method to register this block's tunable dialog parameters as run-time parameters.

**AutoUpdateRuntimePrms**

**Purpose**
Update a block's run-time parameters.

**Syntax**
AutoUpdateRuntimePrms;
Description

Automatically update the values of the run-time parameters during a call to ProcessParameters.

See the S-function `matlabroot/toolbox/simulink/simdemos/simfeatures/adapt_lms.m` in the Simulink model `sldemo_msfcn_lms` for an example.

IsDoingConstantOutput

Purpose

Determine whether this is in the constant sample time stage of a simulation.

Syntax

bVal = IsDoingConstantOutput;

Description

Returns true if this is the constant sample time stage of a simulation, i.e., the stage at the beginning of a simulation where Simulink software computes the values of block outputs that cannot change during the simulation (see “Constant Sample Time”). Use this method in the Outputs method of an S-function with port-based sample times to avoid unnecessarily computing the outputs of ports that have constant sample time, i.e., `[inf, 0]`.

```matlab
function Outputs(block)
    ...
    if block.IsDoingConstantOutput
        ts = block.OutputPort(1).SampleTime;
        if ts(1) == Inf
            % Compute port's output.
            end
        end
    ...
    end
    end
    ...
    end % end of Outputs

See “Specifying Port-Based Sample Times” for more information.
IsMajorTimeStep

**Purpose.**
Determine whether current time step is a major or a minor time step.

**Syntax**
bVal = IsMajorTimeStep;

**Description**
Returns true if the current time step is a major time step; false, if it is a minor time step. This method can be called only from the Outputs or Update methods.

IsSampleHit

**Purpose**
Determine whether the current simulation time is one at which a task handled by this block is active.

**Syntax**
bVal = IsSampleHit(stIdx);

**Arguments**
stIdx
Global index of the sample time to be queried.

**Description**
Use in Outputs or Update block methods when the MATLAB S-function has multiple sample times to determine whether a sample hit has occurred at stIdx. The sample time index stIdx is a global index for the Simulink model. For example, consider a model that contains three sample rates of 0.1, 0.2, and 0.5, and a MATLAB S-function block that contains two rates of 0.2 and 0.5. In the MATLAB S-function, block.IsSampleHit(0) returns true for the rate 0.1, not the rate 0.2.

This block method is similar to ssIsSampleHit for C-Mex S-functions, however ssIsSampleHit returns values based on only the sample times contained in the S-
function. For example, if the model described above contained a C-MeX S-function with sample rates of 0.2 and 0.5, \texttt{ssIsSampleHit(S,0,tid)} returns true for the rate of 0.2.

Use port-based sample times to avoid using the global sample time index for multi-rate systems (see \texttt{Simulink.BlockPortData}).

\textbf{IsSpecialSampleHit}

\textbf{Purpose}

Determine whether the current simulation time is one at which multiple tasks implemented by this block are active.

\textbf{Syntax}

\begin{verbatim}
 bVal = IsSpecialSampleHit(stIdx1,stIdx1);
\end{verbatim}

\textbf{Arguments}

\begin{itemize}
  \item \texttt{stIdx1}  
    Index of sample time of first task to be queried.
  \item \texttt{stIdx2}  
    Index of sample time of second task to be queried.
\end{itemize}

\textbf{Description}

Use in \texttt{Outputs} or \texttt{Update} block methods to ensure the validity of data shared by multiple tasks running at different rates. Returns true if a sample hit has occurred at \texttt{stIdx1} and a sample hit has also occurred at \texttt{stIdx2} in the same time step (similar to \texttt{ssIsSpecialSampleHit} for C-Mex S-functions).

When using the \texttt{IsSpecialSampleHit} macro, the slower sample time must be an integer multiple of the faster sample time.

\textbf{RegBlockMethod}

\textbf{Purpose}

Register a block callback method.
Syntax
RegBlockMethod(methName, methHandle);

Arguments
methName
    Name of method to be registered.
methHandle
    MATLAB function handle of the callback method to be registered.

Description
Registers the block callback method specified by methName and methHandle. Use this method in the setup function of a Level-2 MATLAB S-function to specify the block callback methods that the S-function implements.

RegisterDataTypeFxpBinaryPoint

Purpose
Register fixed-point data type with binary point-only scaling.

Syntax
dtID = RegisterDataTypeFxpBinaryPoint(isSigned, wordLength, fractionalLength, obeyDataTypeOverride);

Arguments
isSigned
    true if the data type is signed.
    false if the data type is unsigned.
wordLength
    Total number of bits in the data type, including any sign bit.
fractionalLength
    Number of bits in the data type to the right of the binary point.
**obeyDataTypeOverride**

true indicates that the Data Type Override setting for the subsystem is to be obeyed. Depending on the value of Data Type Override, the resulting data type could be Double, Single, ScaledDouble, or the fixed-point data type specified by the other arguments of the function.

false indicates that the Data Type Override setting is to be ignored.

**Description**

This method registers a fixed-point data type with Simulink software and returns a data type ID. The data type ID can be used to specify the data types of input and output ports, run-time parameters, and DWork states. It can also be used with all the standard data type access methods defined for instances of this class, such as “DatatypeSize” on page 5-509.

Use this function if you want to register a fixed-point data type with binary point-only scaling. Alternatively, you can use one of the other fixed-point registration functions:

- Use “RegisterDataTypeFxpFSlopeFixexpBias” on page 5-460 to register a data type with [Slope Bias] scaling by specifying the word length, fractional slope, fixed exponent, and bias.
- Use “RegisterDataTypeFxpSlopeBias” on page 5-462 to register a data type with [Slope Bias] scaling.

If the registered data type is not one of the Simulink built-in data types, a Fixed-Point Designer license is checked out.

**RegisterDataTypeFxpFSlopeFixexpBias**

**Purpose**

Register fixed-point data type with [Slope Bias] scaling specified in terms of fractional slope, fixed exponent, and bias

**Syntax**

dtID = RegisterDataTypeFxpFSlopeFixexpBias(isSigned, wordLength, fractionalSlope, fixedExponent, bias, obeyDataTypeOverride);
Arguments

**isSigned**
- true if the data type is signed.
- false if the data type is unsigned.

**wordLength**
- Total number of bits in the data type, including any sign bit.

**fractionalSlope**
- Fractional slope of the data type.

**fixedexponent**
- Exponent of the slope of the data type.

**bias**
- Bias of the scaling of the data type.

**obeyDataTypeOverride**
- true indicates that the Data Type Override setting for the subsystem is to be obeyed. Depending on the value of Data Type Override, the resulting data type could be True Doubles, True Singles, ScaledDouble, or the fixed-point data type specified by the other arguments of the function.

- false indicates that the Data Type Override setting is to be ignored.

Description

This method registers a fixed-point data type with Simulink software and returns a data type ID. The data type ID can be used to specify the data types of input and output ports, run-time parameters, and DWork states. It can also be used with all the standard data type access methods defined for instances of this class, such as “DatatypeSize” on page 5-509.

Use this function if you want to register a fixed-point data type by specifying the word length, fractional slope, fixed exponent, and bias. Alternatively, you can use one of the other fixed-point registration functions:

- Use “RegisterDataTypeFxpBinaryPoint” on page 5-459 to register a data type with binary point-only scaling.
- Use “RegisterDataTypeFxpSlopeBias” on page 5-462 to register a data type with [Slope Bias] scaling.
If the registered data type is not one of the Simulink built-in data types, a Fixed-Point Designer license is checked out.

**RegisterDataTypeFxpSlopeBias**

**Purpose**

Register data type with [Slope Bias] scaling.

**Syntax**

```matlab
dtID = RegisterDataTypeFxpSlopeBias(isSigned, wordLength, totalSlope, bias, obeyDataTypeOverride);
```

**Arguments**

- **isSigned**
  
  true if the data type is signed.
  
  false if the data type is unsigned.

- **wordLength**
  
  Total number of bits in the data type, including any sign bit.

- **totalSlope**
  
  Total slope of the scaling of the data type.

- **bias**
  
  Bias of the scaling of the data type.

- **obeyDataTypeOverride**
  
  true indicates that the **Data Type Override** setting for the subsystem is to be obeyed. Depending on the value of **Data Type Override**, the resulting data type could be True Doubles, True Singles, ScaledDouble, or the fixed-point data type specified by the other arguments of the function.
  
  false indicates that the **Data Type Override** setting is to be ignored.

**Description**

This method registers a fixed-point data type with Simulink software and returns a data type ID. The data type ID can be used to specify the data types of input and output ports,
run-time parameters, and DWork states. It can also be used with all the standard data type access methods defined for instances of this class, such as “DatatypeSize” on page 5-509.

Use this function if you want to register a fixed-point data type with [Slope Bias] scaling. Alternatively, you can use one of the other fixed-point registration functions:

- Use “RegisterDataTypeFxpBinaryPoint” on page 5-459 to register a data type with binary point-only scaling.
- Use “RegisterDataTypeFxpFSlopeFixexpBias” on page 5-460 to register a data type by specifying the word length, fractional slope, fixed exponent, and bias.

If the registered data type is not one of the Simulink built-in data types, a Fixed-Point Designer license is checked out.

**SetAccelRunOnTLC**

**Purpose**

Specify whether to use block's TLC file to generate code for the Accelerator mode of Simulink software.

**Syntax**

```
SetAccelRunOnTLC(bVal);
```

**Arguments**

- **bVal**
  - May be 'true' (use TLC file) or 'false' (run block in interpreted mode).

**Description**

Specify if the block should use its TLC file to generate code that runs with the accelerator. If this option is 'false', the block runs in interpreted mode. See the S-function `msfcn_times_two.m` in the Simulink model `msfcndemo_timestwo` for an example.

**Note** The default JIT Accelerator mode does not support inlining of user-written TLC S-Functions. Please see “How Acceleration Modes Work” and “Control S-Function Execution” for more information.
SetPreCompInpPortInfoToDynamic

Purpose

Set precompiled attributes of this block's input ports to be inherited.

Syntax

SetPreCompInpPortInfoToDynamic;

Description

Initialize the compiled information (dimensions, data type, complexity, and sampling mode) of this block's input ports to be inherited. See the S-function `matlabroot/toolbox/simulink/simdemos/simfeatures/adapt_lms.m` in the Simulink model `sldemo_msfcn_lms` for an example.

SetPreCompOutPortInfoToDynamic

Purpose

Set precompiled attributes of this block's output ports to be inherited.

Syntax

SetPreCompOutPortInfoToDynamic;

Description

Initialize the compiled information (dimensions, data type, complexity, and sampling mode) of the block's output ports to be inherited. See the S-function `matlabroot/toolbox/simulink/simdemos/simfeatures/adapt_lms.m` in the Simulink model `sldemo_msfcn_lms` for an example.

SetPreCompPortInfoToDefaults

Purpose

Set precompiled attributes of this block's ports to the default values.
Syntax

SetPreCompPortInfoToDefaults;

Description

Initialize the compiled information (dimensions, data type, complexity, and sampling mode) of the block's ports to the default values. By default, a port accepts a real scalar sampled signal with a data type of double.

SetSimViewingDevice

Purpose

Specify whether this block is a viewer.

Syntax

SetSimViewingDevice(bVal);

Arguments

bVal

May be 'true' (is a viewer) or 'false' (is not a viewer).

Description

Specify if the block is a viewer/scope. If this flag is specified, the block will be used only during simulation and automatically stubbed out in generated code.

SupportsMultipleExecInstances

Purpose

Specify whether or not a For Each Subsystem supports an S-function inside of it.

Syntax

SupportsMultipleExecInstances(bVal);
Arguments

bVal

May be 'true' (S-function is supported) or 'false' (S-function is not supported).

Description

Specify if an S-function can operate within a For Each Subsystem.

WriteRTWParam

Purpose

Write a custom parameter to the Simulink Coder information file used for code generation.

Syntax

WriteRTWParam(pType, pName, pVal)

Arguments

pType

Type of the parameter to be written. Valid values are 'string' and 'matrix'.

pName

Name of the parameter to be written.

pVal

Value of the parameter to be written.

Description

Use in the WriteRTW method of the MATLAB S-function to write out custom parameters. These parameters are generally settings used to determine how code should be generated in the TLC file for the S-function. See the S-function matlabroot/toolbox/simulink/simdemos/simfeatures/adapt_lms.m in the Simulink model sldemo_msfcn_lms for an example.

Introduced before R2006a
Simulink.NumericType

Specify floating-point, integer, or fixed-point data type

Description

Use a Simulink.NumericType object to set and share numeric data types for signal, state, and parameter data in a model.

1. Create an instance of this class in the MATLAB base workspace, a model workspace, or a data dictionary. To create a numeric type in a model workspace, you must clear the Is alias property.
2. Set the properties of the object to create a custom floating point, integer, or fixed point data type.
3. Assign the data type to all signals and parameters of your model that you want to conform to the data type.

Assigning a data type in this way allows you to change the data types of the signals and parameters in your model by changing the properties of the object that describe them. You do not have to change the model itself.

To rename a data type in a model and in the code that you generate from a model (by generating a typedef statement), you can use an object of the class Simulink.AliasType.

Creation

To create a Simulink.NumericType object interactively, use the Model Explorer:

1. On the Model Explorer Model Hierarchy pane, select a workspace, such as the base workspace, or a data dictionary.
2. From the Model Explorer Add menu, select Simulink.NumericType.

The Model Explorer creates an instance of a Simulink.NumericType object and assigns it to a variable named Numeric in the target workspace.
3 Rename the variable to a more appropriate name, for example, a name that reflects its intended usage.

   To change the name, edit the name displayed in the Name field on the Model Explorer Contents pane.

4 On the Model Explorer Dialog pane, use the Data type mode property to select a data type that the object represents.

   To create a Simulink.NumericType object programmatically, use the Simulink.NumericType function described below.

   **Syntax**

   typeObj = Simulink.NumericType

   **Description**

   typeObj = Simulink.NumericType returns a Simulink.NumericType object with default property values.

   **Properties**

   **Bias — Bias for slope and bias scaling**

   0 (default) | real number

   Bias for slope and bias scaling of a fixed-point data type (Fixed-Point Designer), specified as a real number.

   If you use a number with a data type other than double to set the value, Simulink converts the value to double.

   Corresponds to Bias in the property dialog box.

   Example: 3

   Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | logical
DataScope — Specification to generate or import type definition in the
generated code
'Auto' (default) | 'Exported' | 'Imported'

Specification to generate or import the type definition (typedef) in the generated code
(Simulink Coder), specified as 'Auto', 'Exported, or 'Imported'.

The table shows the effect of each option.

<table>
<thead>
<tr>
<th>Value</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>'Auto' (default)</td>
<td>If no value is specified for HeaderFile, export the type definition to <em>model_types.h</em>. <em>model</em> is the model name. If a value is specified for HeaderFile, import the data type definition from the specified header file.</td>
</tr>
<tr>
<td>'Exported'</td>
<td>Export the data type definition to a header file, which can be specified in the HeaderFile property. If no value is specified for HeaderFile, the header file name defaults to <em>type</em>.h. <em>type</em> is the data type name.</td>
</tr>
<tr>
<td>'Imported'</td>
<td>Import the data type definition from a header file, which can be specified in the HeaderFile property. If no value is specified for HeaderFile, the header file name defaults to <em>type</em>.h. <em>type</em> is the data type name.</td>
</tr>
</tbody>
</table>

For more information, see “Control File Placement of Custom Data Types” (Embedded Coder).

Corresponds to Data scope in the property dialog box.

DataTypeMode — Mode of numeric data type
'Double' (default) | 'Single' | 'Boolean' | 'Fixed-point: unspecified scaling' | 'Fixed-point: binary point scaling' | 'Fixed-point: slope and bias scaling'

Mode of the numeric data type, specified as one of these character vectors:

- 'Double' — Same as the MATLAB double type.
- 'Single' — Same as the MATLAB single type.
- 'Boolean' — Same as the MATLAB boolean type.
- 'Fixed-point: unspecified scaling' — A fixed-point data type with unspecified scaling.
• 'Fixed-point: binary point scaling' — A fixed-point data type with binary-point scaling.
• 'Fixed-point: slope and bias scaling' — A fixed-point data type with slope and bias scaling.

Selecting a fixed-point data type mode can, depending on the other dialog box options that you select, cause the model to run only on systems that have a Fixed-Point Designer option installed.

Corresponds to **Data type mode** in the property dialog box.

**Data Types:** char

**DataTypeOverride — Data type override mode**

'Inherit' (default) | 'Off'

Data type override mode, specified as 'Inherit' or 'Off'.

- If you specify 'Inherit', the data type override setting for the context in which this numeric type is used (block, signal, Stateflow chart in Simulink) applies to this numeric type.
- If you specify 'Off', data type override does not apply to this numeric type.

For more information about data type override, see “Control Data Type Override”.

Corresponds to **Data type override** in the property dialog box.

**Data Types:** char

**Description — Custom description of data type**

' ' (empty character vector) (default) | character vector

Custom description of the data type, specified as a character vector.

Corresponds to **Description** in the property dialog box.

Example: 'This is a floating-point data type.'

**Data Types:** char

**FixedExponent — Exponent for binary point scaling**

0 (default) | real number

Exponent for binary point scaling, specified as a real number. Setting this property causes Simulink software to set the **FractionLength** and **Slope** properties accordingly, and
vice versa. This property applies only if the `DataTypeMode` is `Fixed-point: binary point scaling` or `Fixed-point: slope and bias scaling`.

If you use a number with a data type other than `double` to set the value, Simulink converts the value to `double`.

This property does not appear in the property dialog box.

**Example:** -8

**Data Types:** `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

**FractionLength** — Bit length of the fractional portion of a fixed-point number

0 (default) | real integer

Bit length of the fractional portion of a fixed-point number (Fixed-Point Designer), specified as a real integer. This property equals `-FixedExponent`. Setting this property causes Simulink software to set the `FixedExponent` property accordingly, and vice versa.

If you use a number with a data type other than `double` to set the value, Simulink converts the value to `double`.

Corresponds to **Fraction length** in the property dialog box.

**Example:** 8

**Data Types:** `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

**HeaderFile** — Name of header file that contains type definition in the generated code

'' (empty character vector) (default) | character vector

Name of the header file that contains the type definition (typedef) in the generated code, specified as a character vector.

If this property is specified, the specified name is used during code generation for importing or exporting. If this property is empty, the value defaults to `type.h` if `DataScope` equals `'Imported'` or `'Exported'`, or defaults to `model_types.h` if `DataScope` equals `'Auto'`. 
By default, the generated `#include` directive uses the preprocessor delimiter " instead of `< and >`. To generate the directive `#include <myTypes.h>`, specify `HeaderFile` as '<myTypes.h>'.

For more information, see “Control File Placement of Custom Data Types” (Embedded Coder).

Corresponds to **Header file** in the property dialog box.

Example: 'myHdr.h'

Example: 'myHdr'

Example: 'myHdr.hpp'

Data Types: char

**IsAlias — Specification to create data type alias using object name**

false (default) | true

Specification to create a data type alias by using the name of the object, specified as true (yes) or false (no).

If you specify true, the object acts as a data type alias in a similar manner to a `Simulink.AliasType` object. For more information, see “Control Data Type Names in Generated Code” (Embedded Coder).

Corresponds to **Is alias** in the property dialog box.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | logical

**Signedness — Signedness of fixed-point data type**

'Signed (default) | 'Unsigned' | 'Auto'

Signedness of a fixed-point data type (Fixed-Point Designer), specified as 'Signed' (signed), 'Unsigned' (unsigned), or 'Auto' (inherit signedness).

Corresponds to **Signedness** in the property dialog box.

Data Types: char

**SignednessBool — Signedness of fixed-point data type**

false | true

5-472
Signedness of a fixed-point data type (Fixed-Point Designer), specified as a Boolean.

Corresponds to Signedness in the property dialog box.

Data Types: logical

**Slope — Slope for slope and bias scaling of fixed-point data type**

2^0 (default) | real number

Slope for slope and bias scaling of a fixed-point data type (Fixed-Point Designer), specified as a real number.

This property equals SlopeAdjustmentFactor * 2^FixedExponent. If SlopeAdjustmentFactor is 1.0, Simulink software displays the value of this field as 2^SlopeAdjustmentFactor. Otherwise, it displays it as a numeric value. Setting this property causes Simulink software to set the FixedExponent and SlopeAdjustmentFactor properties accordingly, and vice versa.

If you use a number with a data type other than double to set the value, Simulink converts the value to double.

This property appears only if DataTypeMode is Fixed-point: slope and bias scaling.

Corresponds to Slope in the property dialog box.

Example: 5.2
Example: 2^9

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | logical

**SlopeAdjustmentFactor — Slope for slope and bias scaling of fixed-point data type**

1 (default) | real number

Slope for slope and bias scaling of a fixed-point data type (Fixed-Point Designer), specified as a real number in the range [1, 2).

Setting this property causes Simulink software to adjust the Slope property accordingly, and vice versa. This property applies only if DataTypeMode is Fixed-point: slope and bias scaling.
If you use a number with a data type other than `double` to set the value, Simulink converts the value to `double`.

This property does not appear in the property dialog box.

Example: `1.7`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

**WordLength — Word size of fixed-point or integer data type**

16 (default) | integer

Word size of a fixed-point (Fixed-Point Designer) or integer data type, specified as an integer number of bits.

This property appears only if `DataTypeMode` is `Fixed-point`.

If you use a number with a data type other than `double` to set the value, Simulink converts the value to `double`.

Corresponds to **Word length** in the property dialog box.

Example: `8`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

**Object Functions**

- `isboolean` Determine whether numeric type represents the Boolean data type boolean
- `isdouble` Determine whether numeric type represents the double-precision, floating-point data type double
- `isfixed` Determine whether numeric type represents a fixed-point data type
- `isfloat` Determine whether numeric type represents a floating-point data type
- `isscalingbinarypoint` Determine whether fixed-point numeric type has binary-point scaling
- `isscalingslopebias` Determine whether numeric type represents a fixed-point data type with slope-and-bias scaling
isscalingunspecified  Determine whether numeric type represents a data type with unspecified scaling
issingle  Determine whether numeric type represents the single-precision, floating-point data type single

Examples

Share a Data Type Between Separate Algorithms, Data Paths, Models, and Bus Elements

See “Share a Data Type Between Separate Algorithms, Data Paths, Models, and Bus Elements”.

See Also

Simulink.AliasType

Topics

“Validate a Floating-Point Embedded Model”
“Control Signal Data Types”
“Control Data Type Names in Generated Code” (Embedded Coder)
“Data Types Supported by Simulink”
“About Data Types in Simulink”

Introduced before R2006a
isboolean

Package: Simulink

Determine whether numeric type represents the Boolean data type boolean

Syntax

indication = isboolean(numericType)

Description

indication = isboolean(numericType) returns 1 (true) if the Simulink.NumericType object numericType represents the Boolean data type boolean and 0 (false) otherwise.

In Simulink, a Simulink.NumericType object represents a data type that you can share between different data items in a model. For more information, see Simulink.NumericType.

Input Arguments

numericType — Target numeric type
Simulink.NumericType object

Target numeric type, specified as a Simulink.NumericType object.

Output Arguments

indication — Indication of whether target object represents boolean
1 | 0

Indication of whether the target object represents boolean, returned as 1 (true) or 0 (false).
See Also
Simulink.NumericType

Topics
“Control Signal Data Types”
“Control Block Parameter Data Types”
“Control Data Type Names in Generated Code” (Embedded Coder)

Introduced in R2010b
**isdoubles**

**Package:** Simulink

Determine whether numeric type represents the double-precision, floating-point data type double

**Syntax**

```
indication = isdouble(numericType)
```

**Description**

```
indication = isdouble(numericType) returns 1 (true) if the
Simulink.NumericType object numericType represents the double-precision, floating-
point data type double and 0 (false) otherwise.
```

In Simulink, a Simulink.NumericType object represents a data type that you can share
between different data items in a model. For more information, see
Simulink.NumericType.

**Input Arguments**

- **numericType** — Target numeric type
  
  Simulink.NumericType object

  Target numeric type, specified as a Simulink.NumericType object.

**Output Arguments**

- **indication** — Indication of whether target object represents double
  
  1 | 0

  Indication of whether the target object represents double, returned as 1 (true) or 0 (false).
See Also
Simulink.NumericType | isfloat | issingle

Topics
“Control Signal Data Types”
“Control Block Parameter Data Types”
“Control Data Type Names in Generated Code” (Embedded Coder)

Introduced in R2010b

isfixed

Package: Simulink

Determine whether numeric type represents a fixed-point data type

Syntax

indication = isfixed(numericType)

Description

indication = isfixed(numericType) returns 1 (true) if the Simulink.NumericType object numericType represents a fixed-point data type and 0 (false) otherwise.

In Simulink, a Simulink.NumericType object represents a data type that you can share between different data items in a model. For more information, see Simulink.NumericType.

Input Arguments

numericType — Target numeric type
Simulink.NumericType object

Target numeric type, specified as a Simulink.NumericType object.

Output Arguments

indication — Indication of whether target object represents a fixed-point type
1 | 0

Indication of whether the target object represents a fixed-point type, returned as 1 (true) or 0 (false).
See Also
Simulink.NumericType | isscalingbinarypoint | isscalingslopebias | isscalingunspecified

Topics
“Control Signal Data Types”
“Control Block Parameter Data Types”
“Control Data Type Names in Generated Code” (Embedded Coder)

Introduced in R2010b
isfloat

Package: Simulink

Determine whether numeric type represents a floating-point data type

Syntax

indication = isfloat(numericType)

Description

indication = isfloat(numericType) returns 1 (true) if the Simulink.NumericType object numericType represents a floating-point data type such as double or single, and 0 (false) otherwise.

In Simulink, a Simulink.NumericType object represents a data type that you can share between different data items in a model. For more information, see Simulink.NumericType.

Input Arguments

numericType — Target numeric type
Simulink.NumericType object

Target numeric type, specified as a Simulink.NumericType object.

Output Arguments

indication — Indication of whether target object represents a floating-point type
1 | 0

Indication of whether the target object represents a floating-point type, returned as 1 (true) or 0 (false).
See Also
Simulink.NumericType | isdouble | issingle

Topics
“Control Signal Data Types”
“Control Block Parameter Data Types”
“Control Data Type Names in Generated Code” (Embedded Coder)

Introduced in R2010b
isscalingbinarypoint

Package: Simulink

Determine whether fixed-point numeric type has binary-point scaling

Syntax

indication = isscalingbinarypoint(numericType)

Description

indication = isscalingbinarypoint(numericType) returns 1 (true) if the Simulink.NumericType object numericType represents a fixed-point data type with binary-point scaling and 0 (false) otherwise. A numeric type object can use binary-point scaling if you explicitly specify it or if you specify trivial slope-and-bias scaling (the slope is an integer power of two and the bias is zero).

In Simulink, a Simulink.NumericType object represents a data type that you can share between different data items in a model. For more information, see Simulink.NumericType.

Input Arguments

numericType — Target numeric type
Simulink.NumericType object

Target numeric type, specified as a Simulink.NumericType object.

Output Arguments

indication — Indication of whether target object represents fixed-point type with binary-point scaling
1 | 0
Indication of whether the target object represents a fixed-point type with binary-point scaling, returned as `1` (true) or `0` (false).

**See Also**
Simulink.NumericType | isfixed | isscalingslopebias | isscalingunspecified

**Topics**
“Control Signal Data Types”
“Control Block Parameter Data Types”
“Control Data Type Names in Generated Code” (Embedded Coder)

**Introduced in R2010b**
isscalingslopebias

Package: Simulink

Determine whether numeric type represents a fixed-point data type with slope-and-bias scaling

Syntax

indication = isscalingslopebias(numericType)

Description

indication = isscalingslopebias(numericType) returns 1 (true) if the Simulink.NumericType object numericType represents a fixed-point data type with nontrivial slope-and-bias scaling and 0 (false) otherwise. A slope-and-bias fixed-point type has trivial scaling if the slope is an integer power of two and the bias is zero.

In Simulink, a Simulink.NumericType object represents a data type that you can share between different data items in a model. For more information, see Simulink.NumericType.

Input Arguments

numericType — Target numeric type
Simulink.NumericType object

Target numeric type, specified as a Simulink.NumericType object.

Output Arguments

indication — Indication of whether target object represents fixed-point type with slope-and-bias scaling
1 | 0
Indication of whether the target object represents a fixed-point type with nontrivial slope-and-bias scaling, returned as 1 (true) or 0 (false).

**See Also**
Simulink.NumericType | isfixed | isscalingbinarypoint | isscalingunspecified

**Topics**
“Control Signal Data Types”
“Control Block Parameter Data Types”
“Control Data Type Names in Generated Code” (Embedded Coder)

**Introduced in R2010b**
isscalingunspecified

Package: Simulink

Determine whether numeric type represents a data type with unspecified scaling

Syntax

indication = isscalingunspecified(numericType)

Description

indication = isscalingunspecified(numericType) returns 1 (true) if the Simulink.NumericType object numericType represents a fixed-point or scaled double data type with unspecified scaling and 0 (false) otherwise.

In Simulink, a Simulink.NumericType object represents a data type that you can share between different data items in a model. For more information, see Simulink.NumericType.

Input Arguments

numericType — Target numeric type
Simulink.NumericType object

Target numeric type, specified as a Simulink.NumericType object.

Output Arguments

indication — Indication of whether target object represents a type with unspecified scaling
1 | 0

Indication of whether the target object represents a type with unspecified scaling, returned as 1 (true) or 0 (false).
See Also
Simulink.NumericType | isfixed | isscalingbinarypoint |
isscalingslopebias

Topics
“Control Signal Data Types”
“Control Block Parameter Data Types”
“Control Data Type Names in Generated Code” (Embedded Coder)

Introduced in R2010b
issingle

Package: Simulink

Determine whether numeric type represents the single-precision, floating-point data type single.

Syntax

indication = issingle(numericType)

Description

indication = issingle(numericType) returns 1 (true) if the Simulink.NumericType object numericType represents the single-precision, floating-point data type single and 0 (false) otherwise.

In Simulink, a Simulink.NumericType object represents a data type that you can share between different data items in a model. For more information, see Simulink.NumericType.

Input Arguments

numericType — Target numeric type
Simulink.NumericType object

Target numeric type, specified as a Simulink.NumericType object.

Output Arguments

indication — Indication of whether target object represents single
1 | 0

Indication of whether the target object represents single, returned as 1 (true) or 0 (false).
See Also
Simulink.NumericType | isdouble | isfloat

Topics
“Control Signal Data Types”
“Control Block Parameter Data Types”
“Control Data Type Names in Generated Code” (Embedded Coder)

Introduced in R2010b
Simulink.Parameter

Store, share, and configure block parameter values

Description

Create a Simulink.Parameter object to set the value of one or more block parameters in a model, such as the Gain parameter of a Gain block. You create the object in a workspace or in a data dictionary. Set the parameter value in the object, not in the block.

Use a Simulink.Parameter object to:

- Share a value among multiple block parameters.
- Represent an engineering constant or a tunable calibration parameter.
- Separate a parameter value from its data type.
- Configure parameter data for code generation.

The Value property of the object stores the parameter value. To use the object in a model, set the value of a block parameter to an expression that involves the name of the object. Omit the Value property from the expression. For more information, see “Use Parameter Objects”.

For more information about block parameters, see “Set Block Parameter Values” and “How Generated Code Stores Internal Signal, State, and Parameter Data” (Simulink Coder).

Creation

Create a Simulink.Parameter object:

- Directly from a block dialog box or the Property Inspector. See “Create, Edit, and Manage Workspace Variables”.
- By using the Model Data Editor. Inspect the Parameters tab.
- By using the Model Explorer. See “Create Data Objects from Built-In Data Class Package Simulink”.

5-492
• By using the Simulink.Parameter function, described below.

**Syntax**

```matlab
paramObj = Simulink.Parameter
paramObj = Simulink.Parameter(paramValue)
```

**Description**

`paramObj = Simulink.Parameter` returns a `Simulink.Parameter` object with default property values.

`paramObj = Simulink.Parameter(paramValue)` returns a `Simulink.Parameter` object and initializes the value of the `Value` property by using `paramValue`.

**Properties**

For information about properties in the property dialog box of a `Simulink.Parameter` object, see “Simulink.Parameter Property Dialog Box”.

**CoderInfo — Specifications for generating code for parameter object**

`Simulink.CoderInfo` object

Specifications for generating code for the parameter object, returned as a `Simulink.CoderInfo` object.

This property is read only. Instead, modify the properties of the `Simulink.CoderInfo` object that this property contains.

For example, the `StorageClass` property of the `Simulink.CoderInfo` object determines how Simulink code generation toolboxes allocate memory for the parameter object in the generated code. For more information, see “Apply Storage Classes to Individual Signal, State, and Parameter Data Elements” (Simulink Coder) and “Choose Storage Class for Controlling Data Representation in Generated Code” (Embedded Coder).

**Complexity — Numeric complexity of parameter value**

'`real`' (default) | 'complex'
Numeric complexity of the parameter value, returned as 'real' (if the value is real) or 'complex' (if the value is complex). Simulink determines the complexity from the parameter value that you specify in the Value property. This property is read only.

Data Types: char

**DataType — Data type of parameter value**

'REAL' (default) | character vector

Data type of the parameter value that you specify in the Value property. When you simulate the model or generate code, Simulink casts the value to the specified data type.

If you specify 'auto', the default setting, the parameter object uses the same data type as the block parameters that use the object. See “Reduce Maintenance Effort with Data Type Inheritance”.

When you set the Value property by using something other than a double number, the object typically sets the DataType property based on the value of the Value property. For example, when you set the Value property to int8(5), the object sets the value of the DataType property to 'int8'.

To explicitly specify a built-in data type (see “Data Types Supported by Simulink”), specify one of these options:

- 'double'
- 'single'
- 'int8'
- 'uint8'
- 'int16'
- 'uint16'
- 'int32'
- 'uint32'
- 'int64'
- 'uint64'
- 'boolean'

To specify a fixed-point data type, use the fixdt function. For example, specify 'fixdt(1,16,5)'.

5-494
If you use a Simulink.AliasType or Simulink.NumericType object to create and share custom data types in your model, specify the name of the object.

To specify an enumerated data type, use the name of the type preceded by Enum:. For example, specify 'Enum: myEnumType'.

When you store a structure or array of structures in the Value property of the object, the object sets the DataType property to 'struct'. To specify a Simulink.Bus object as the data type, use the name of the bus object preceded by Bus:. For example, specify 'Bus: myBusObject'.

Example: 'auto'
Example: 'int8'
Example: 'fixdt(1,16,5)'
Example: 'myAliasTypeObject'
Example: 'Enum: myEnumType'
Example: 'Bus: myBusObject'

Data Types: char

Description — Custom description of parameter object

' ' (empty character vector) (default) | character vector

Custom description of the parameter object, specified as a character vector. Use this property to document the significance that the parameter object has in your algorithm.

If you have Embedded Coder, you can configure this description to appear in the generated code as a comment. See “Simulink data object descriptions” (Simulink Coder).

Example: 'This parameter represents the maximum rotation speed of the engine.'

Data Types: char

Dimensions — Dimensions of parameter value

[0 0] (default) | row vector | character vector

Dimensions of the value stored in the Value property, returned as a row vector or specified as a character vector.
When you set the `Value` property of the object, the object sets the value of the `Dimensions` property to a `double` row vector. The vector is the same vector that the `size` function returns.

To use symbolic dimensions, specify a character vector. See “Implement Dimension Variants for Array Sizes in Generated Code” (Embedded Coder).

Example: `[1 3]`
Example: `'[1 myDimParam]'`
Data Types: `double` | `char`

**Max — Maximum value of parameter**

`[]` (empty) (default) | real `double` scalar

Maximum value that the `Value` property of the object can store, specified as a real `double` scalar.

The default value is `[]` (empty), which means the parameter value does not have a maximum.

If you store a complex number in the `Value` property, the `Max` property applies separately to the real and imaginary parts.

If you store a structure in the `Value` property, the object ignores the `Max` property. Instead, use a `Simulink.Bus` object as the data type of the parameter object, and specify a maximum value for each field by using the elements of the bus object. See “Control Field Data Types and Characteristics by Creating Parameter Object”.

If the parameter value is greater than the maximum value or if the maximum value is outside the range of the object data type, Simulink generates a warning. When updating the diagram or starting a simulation, Simulink generates an error.

For more information about how Simulink uses this property, see “Specify Minimum and Maximum Values for Block Parameters”.

Example: 5.32
Data Types: `double`

**Min — Minimum value of parameter**

`[]` (empty) (default) | real `double` scalar
Minimum value that the Value property of the object can store, specified as a real double scalar.

The default value is [] (empty), which means the parameter value does not have a minimum.

If you store a complex number in the Value property, the Min property applies separately to the real and imaginary parts.

If you store a structure in the Value property, the object ignores the Min property. Instead, use a Simulink.Bus object as the data type of the parameter object, and specify a minimum value for each field by using the elements of the bus object. See “Control Field Data Types and Characteristics by Creating Parameter Object”.

If the parameter value is less than the minimum value or if the minimum value is outside the range of the object data type, Simulink generates a warning. When updating the diagram or starting a simulation, Simulink generates an error.

For more information about how Simulink uses this property, see “Specify Minimum and Maximum Values for Block Parameters”

Example: -0.92

Data Types: double

**Unit — Physical unit of parameter value**

empty (default) | valid unit

Physical unit of parameter value, specified as a character vector. For more information, see “Unit Specification in Simulink Models”.

Example: 'degC'

Data Types: char

**Value — Value to use in target block parameters**

[] (default) | valid value

Value to use in target block parameters, specified as any of these valid values:

- Numeric value
- Boolean value
- Instance of enumerated type
• Structure
• Scalar or array
• Mathematical expression (see “Set Variable Value by Using a Mathematical Expression”)

You can use MATLAB syntax to specify the value.

<table>
<thead>
<tr>
<th>Example Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>15.23</td>
<td>Specifies a scalar value</td>
</tr>
<tr>
<td>[3 4; 9 8]</td>
<td>Specifies a matrix</td>
</tr>
<tr>
<td>3+2i</td>
<td>Specifies a complex value</td>
</tr>
<tr>
<td>struct('A',20,'B',5)</td>
<td>Specifies a structure with two fields, A and B, with double-precision values 20 and 5. Organize block parameters into structures (see “Organize Related Block Parameter Definitions in Structures”) or initialize the signal elements in a bus (see “Specify Initial Conditions for Bus Signals”).</td>
</tr>
<tr>
<td>slexpr('myVar + myOtherVar')</td>
<td>Specifies the expression myVar + myOtherVar where myVar and myOtherVar are other MATLAB variables or parameter objects. Simulink preserves this mathematical relationship between the object and the variables.</td>
</tr>
</tbody>
</table>

To use a Simulink.Parameter object to store a value of a particular numeric data type, specify the ideal value with the Value property, and control the type with the DataType property.

If you set the Value property by using a typed expression such as single(32.5), the DataType property changes to reflect the new type. A best practice is using an expression that is not typed. You can avoid accumulating numerical error through repeated quantizations or data type saturation, especially for fixed-point data types.

Example: 3.15
Example: single([3.15 1.23])
Example: 1.2 + 3.2i
Example: true
Example: myEnumType.myEnumValue
Example: struct('field1',15,'field2',7.32)
Example: slexpr('myVar + myOtherVar')

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | logical | struct | fi
Complex Number Support: Yes

Examples

Use Parameter Object to Set Value of Gain Parameter

1  At the command prompt, create a Simulink.Parameter object.
   
   myParam = Simulink.Parameter;

2  Assign a numeric value to the Value property.
   
   myParam.Value = 15.23;

3  Specify other characteristics for the block parameter by adjusting the object
    properties. For example, to specify the minimum and maximum values the parameter
    can take, use the Min and Max properties.

   
   myParam.Min = 10.11;
   myParam.Max = 25.27;

4  In a block dialog box, specify the value of a parameter as myParam. For example, in a
    Gain block dialog box, specify Gain as myParam.

   During simulation, the Gain parameter uses the value 15.23.

Change Value Stored by Parameter Object

1  At the command prompt, create a Simulink.Parameter object that stores the value
   2.52.

   myParam = Simulink.Parameter(2.52);

2  Change the value by accessing the Value property of the object. This technique
   preserves the values of the other properties of the object.

   myParam.Value = 1.13;
Create Parameter Object with Specific Numeric Data Type

To reduce model maintenance, you can leave the DataType property at its default value, auto. The parameter object acquires a data type from the block parameter that uses the object.

To reduce the risk of the data type changing when you make changes to signal data types and other data types in your model, you can explicitly specify a data type for the parameter object. For example, when you generate code that exports parameter data to your custom code, explicitly specify a data type for the object.

1 At the command prompt, create a Simulink.Parameter object that stores the value 18.25.

myParam = Simulink.Parameter(18.25);

The expression 18.25 returns the number 18.25 with the double-precision, floating-point data type double. The Value property stores the number 18.25 with double precision.

2 Use the DataType property to specify the single-precision data type single.

myParam.DataType = 'single';

When you simulate or generate code, the parameter object casts the value of the Value property, 18.25, to the data type specified by the DataType property, single.

Set Parameter Value to a Mathematical Expression

This example shows how to set the value of a parameter object, myParam, to the sum of two other variables, myVar and myOtherVar. With this technique, when you change the values of the independent variables, Simulink immediately calculates the new value of the parameter object.

1 Create the two independent variables.

myVar = 5.2;
myOtherVar = 9.8;

2 Create the parameter object.

myParam = Simulink.Parameter;
3 Set the value of the parameter object to the expression `myVar + myOtherVar`.

```matlab
myParam.Value = slexpr('myVar + myOtherVar')
```

See Also

AUTOSAR.Parameter | Simulink.CoderInfo | Simulink.LookupTable | Simulink.Signal

Topics

“Data Objects”
“Set Block Parameter Values”
“How Generated Code Stores Internal Signal, State, and Parameter Data” (Simulink Coder)
“Determine Where to Store Variables and Objects for Simulink Models”
“Data Types Supported by Simulink”
“Define Data Classes”

Introduced before R2006a
Simulink.RunTimeBlock

Allow Level-2 MATLAB S-function and other MATLAB programs to get information about block while simulation is running

Description

This class allows a Level-2 MATLAB S-function or other MATLAB program to obtain information about a block. Simulink software creates an instance of this class or a derived class for each block in a model. Simulink software passes the object to the callback methods of Level-2 MATLAB S-functions when it updates or simulates a model, allowing the callback methods to get block-related information from and provide such information to Simulink software. See “Write Level-2 MATLAB S-Functions” in Writing S-Functions for more information. You can also use instances of this class in MATLAB programs to obtain information about blocks during a simulation. See “Access Block Data During Simulation” for more information.

Note Simulink.RunTimeBlock objects do not support MATLAB sparse matrices. For example, the following line of code attempts to assign a sparse identity matrix to the runtime object's output port data. This line of code in a Level-2 MATLAB S-function produces an error:

```
block.Outport(1).Data = speye(10);
```

Parent Class

None

Derived Classes

Simulink.MSFcnRunTimeBlock
### Property Summary

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;BlockHandle&quot; on page 5-504</td>
<td>Block's handle.</td>
</tr>
<tr>
<td>&quot;CurrentTime&quot; on page 5-504</td>
<td>Current simulation time.</td>
</tr>
<tr>
<td>&quot;NumDworks&quot; on page 5-505</td>
<td>Number of discrete work vectors used by the block.</td>
</tr>
<tr>
<td>&quot;NumOutputPorts&quot; on page 5-505</td>
<td>Number of block output ports.</td>
</tr>
<tr>
<td>&quot;NumContStates&quot; on page 5-505</td>
<td>Number of block's continuous states.</td>
</tr>
<tr>
<td>&quot;NumDworkDiscStates&quot; on page 5-506</td>
<td>Number of block's discrete states</td>
</tr>
<tr>
<td>&quot;NumDialogPrms&quot; on page 5-506</td>
<td>Number of parameters that can be entered on S-function block's dialog box.</td>
</tr>
<tr>
<td>&quot;NumInputPorts&quot; on page 5-506</td>
<td>Number of block's input ports.</td>
</tr>
<tr>
<td>&quot;NumRuntimePrms&quot; on page 5-507</td>
<td>Number of run-time parameters used by block.</td>
</tr>
<tr>
<td>&quot;SampleTimes&quot; on page 5-507</td>
<td>Sample times at which block produces outputs.</td>
</tr>
</tbody>
</table>

### Method Summary

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;ContStates&quot; on page 5-507</td>
<td>Get a block's continuous states.</td>
</tr>
<tr>
<td>&quot;DataTypeIsFixedPoint&quot; on page 5-508</td>
<td>Determine whether a data type is fixed point.</td>
</tr>
<tr>
<td>&quot;DatatypeName&quot; on page 5-508</td>
<td>Get name of a data type supported by this block.</td>
</tr>
<tr>
<td>&quot;DatatypeSize&quot; on page 5-509</td>
<td>Get size of a data type supported by this block.</td>
</tr>
<tr>
<td>&quot;Derivatives&quot; on page 5-509</td>
<td>Get a block's continuous state derivatives.</td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>------</td>
<td>-------------</td>
</tr>
<tr>
<td>“DialogPrm” on page 5-510</td>
<td>Get a parameter entered on an S-function block's dialog box.</td>
</tr>
<tr>
<td>“Dwork” on page 5-510</td>
<td>Get one of a block's DWork vectors.</td>
</tr>
<tr>
<td>“FixedPointNumericType” on page 5-511</td>
<td>Determine the properties of a fixed-point data type.</td>
</tr>
<tr>
<td>“InputPort” on page 5-512</td>
<td>Get one of a block's input ports.</td>
</tr>
<tr>
<td>“OutputPort” on page 5-512</td>
<td>Get one of a block's output ports.</td>
</tr>
<tr>
<td>“RuntimePrm” on page 5-513</td>
<td>Get one of the run-time parameters used by a block.</td>
</tr>
</tbody>
</table>

### Properties

**BlockHandle**

**Description**

Block's handle.

**Access**

R0

**CurrentTime**

**Description**

Current simulation time.

**Access**

R0
**NumDworks**

**Description**

Number of data work vectors.

**Access**

RW

**See Also**

ssGetNumDWork

**NumOutputPorts**

**Description**

Number of output ports.

**Access**

RW

**See Also**

ssGetNumOutputPorts

**NumContStates**

**Description**

Number of continuous states.

**Access**

RW

**See Also**

ssGetNumContStates
**NumDworkDiscStates**

**Description**

Number of discrete states. In a MATLAB S-function, you need to use DWorks to set up discrete states.

**Access**

RW

**See Also**

ssGetNumDiscStates

**NumDialogPrms**

**Description**

Number of parameters declared on the block's dialog. In the case of the S-function, it returns the number of parameters listed as a comma-separated list in the **S-function parameters** dialog field.

**Access**

RW

**See Also**

ssGetNumSFcnParams

**NumInputPorts**

**Description**

Number of input ports.

**Access**

RW
See Also
ssGetNumInputPorts

NumRuntimePrms
Description
Number of run-time parameters used by this block. See “Create and Update S-Function Run-Time Parameters” for more information.

Access
RW

See Also
ssGetNumSFcnParams

SampleTimes
Description
Block's sample times.

Access
RW for MATLAB S-functions, R0 for all other blocks.

Methods
ContStates
Purpose
Get a block's continuous states.

Syntax
states = ContStates();
**Description**

Get vector of continuous states.

**See Also**

ssGetContStates

**DataTypeIsFixedPoint**

**Purpose**

Determine whether a data type is fixed point.

**Syntax**

```matlab
bVal = DataTypeIsFixedPoint(dtID);
```

**Arguments**

*dtID*

- Integer value specifying the ID of a data type.

**Description**

Returns `true` if the specified data type is a fixed-point data type.

**DatatypeName**

**Purpose**

Get the name of a data type.

**Syntax**

```matlab
name = DatatypeName(dtID);
```

**Arguments**

*dtID*

- Integer value specifying ID of a data type.
**Description**

Returns the name of the data type specified by dtID.

**See Also**

“DatatypeSize” on page 5-509

**DatatypeSize**

**Purpose**

Get the size of a data type.

**Syntax**

```matlab
size = DatatypeSize(dtID);
```

**Arguments**

`dtID`

Integer value specifying the ID of a data type.

**Description**

Returns the size of the data type specified by dtID.

**See Also**

“DatatypeName” on page 5-508

**Derivatives**

**Purpose**

Get derivatives of a block's continuous states.

**Syntax**

```matlab
derivs = Derivatives();
```
**Description**
Get vector of state derivatives.

**See Also**
ssGetdX

**DialogPrm**

**Purpose**
Get an S-function's dialog parameters.

**Syntax**

```matlab
param = DialogPrm(pIdx);
```

**Arguments**

- **pIdx**
  
  Integer value specifying the index of the parameter to be returned.

**Description**

Get the specified dialog parameter. In the case of the S-function, each `DialogPrm` corresponds to one of the elements in the comma-separated list of parameters in the **S-function parameters** dialog field.

**See Also**

ssGetSFcnParam, “RuntimePrm” on page 5-513

**Dwork**

**Purpose**
Get one of a block's DWork vectors.

**Syntax**

```matlab
dworkObj = Dwork(dwIdx);
```
Arguments
dwIdx
  Integer value specifying the index of a work vector.

Description
Get information about the DWork vector specified by dwIdx where dwIdx is the index number of the work vector. This method returns an object of type Simulink.BlockCompDworkData.

See Also
ssGetDWork

FixedPointNumericType

Purpose
Get the properties of a fixed-point data type.

Syntax
eno = FixedPointNumericType(dtID);

Arguments
dtID
  Integer value specifying the ID of a fixed-point data type.

Description
Returns an object of embedded.Numeric class that contains the attributes of the specified fixed-point data type.

Note  embedded.Numeric is also the class of the numerictype objects created by Fixed-Point Designer software. For information on the properties defined by embedded.Numeric class, see “numerictype Object Properties” (Fixed-Point Designer).
**InputPort**

**Purpose**
Get an input port of a block.

**Syntax**

```matlab
port = InputPort(pIdx);
```

**Arguments**

- `pIdx`  
  Integer value specifying the index of an input port.

**Description**
Get the input port specified by `pIdx`, where `pIdx` is the index number of the input port. For example,

```matlab
port = rto.InputPort(1)
```

returns the first input port of the block represented by the run-time object `rto`.

This method returns an object of type `Simulink.BlockPreCompInputPortData` or `Simulink.BlockCompInputPortData`, depending on whether the model that contains the port is uncompiled or compiled. You can use this object to get and set the input port's uncompiled or compiled properties, respectively.

**See Also**

- `ssGetInputPortSignalPtrs`, `Simulink.BlockPreCompInputPortData`, `Simulink.BlockCompInputPortData`, “OutputPort” on page 5-512

**OutputPort**

**Purpose**
Get an output port of a block.

**Syntax**

```matlab
port = OutputPort(pIdx);
```
Arguments

pIdx

Integer value specifying the index of an output port.

Description

Get the output port specified by pIdx, where pIdx is the index number of the output port. For example,

```matlab
port = rto.OutputPort(1)
```

returns the first output port of the block represented by the run-time object rto.

This method returns an object of type `Simulink.BlockPreCompOutputPortData` or `Simulink.BlockCompOutputPortData`, depending on whether the model that contains the port is uncompiled or compiled, respectively. You can use this object to get and set the output port's uncompiled or compiled properties, respectively.

See Also

`ssGetInputPortSignalPtrs`, `Simulink.BlockPreCompOutputPortData`, `Simulink.BlockCompOutputPortData`

RuntimePrm

Purpose

Get an S-function's run-time parameters.

Syntax

```matlab
param = RuntimePrm(pIdx);
```

Arguments

pIdx

Integer value specifying the index of a run-time parameter.
Description

Get the run-time parameter whose index is pIdx. This run-time parameter is a Simulink.BlockData on page 5-220 object of type Simulink.BlockRunTimePrmData.

See Also

ssGetRunTimeParamInfo

Introduced before R2006a
Simulink.SampleTime class

Package: Simulink

Object containing sample time information

Description

The SampleTime class represents the sample time information associated with an individual sample time.

Use the methods Simulink.Block.getSampleTimes and Simulink.BlockDiagram.getSampleTimes to retrieve the values of the SampleTime properties for a block and for a block diagram, respectively.

Properties

Value

A two-element array of doubles that contains the period and offset of the sample time

Description

A 1xn character array that describes the sample time type

ColorRGBValue

A 1x3 array of doubles that contains the red, green and blue (RGB) values of the sample time color

Annotation

A 1xn character array that represents the annotation of a specific sample time (for example, 'D1')
**OwnerBlock**

For asynchronous and variable sample times, `OwnerBlock` is a character vector containing the full path to the block that controls the sample time. For all other types of sample times, it is an empty character vector.

**ComponentSampleTimes**

If the sample time is an async union or if the sample time is hybrid and the component sample times are available, then the array `ComponentSampleTimes` contains `Simulink.SampleTime` objects.

**Copy Semantics**

Value. To learn how value classes affect copy operations, see Copying Objects (MATLAB) in the MATLAB Programming Fundamentals documentation.

**Examples**

Retrieve the sample time information for the 'vdp' model.

```matlab
ts = Simulink.BlockDiagram.getSampleTimes('vdp')
```

Simulink returns:

```matlab
ts =
  1x2 Simulink.SampleTime
Package: Simulink

Properties:
  Value
  Description
  ColorRGBValue
  Annotation
  OwnerBlock
  ComponentSampleTimes

Methods
```

To examine the values of the properties:
ts(1), ts(2)

ans =

Simulink.SampleTime
Package: Simulink

Properties:
  Value: [0 0]
  Description: 'Continuous'
  ColorRGBValue: [0 0 0]
  Annotation: 'Cont'
  OwnerBlock: []
  ComponentSampleTimes: {}

Methods

ans =

Simulink.SampleTime
Package: Simulink

Properties:
  Value: [Inf 0]
  Description: 'Constant'
  ColorRGBValue: [1 0.2600 0.8200]
  Annotation: 'Inf'
  OwnerBlock: []
  ComponentSampleTimes: {}

Methods

See Also
Simulink.Block.getSampleTimes | Simulink.BlockDiagram.getSampleTimes
Simulink.sdi.constraints.MatchesSignal class

**Package:** Simulink.sdi.constraints  
**Superclasses:** matlab.unittest.constraints.Constraint

Constraint that compares time series data with tolerances using the Simulation Data Inspector

**Description**

Use the `Simulink.sdi.constraints.MatchesSignal` constraint to compare time series data against an expected value as part of a test. You can configure the constraint to use a combination of absolute, relative, and time tolerance values for the comparison. You can also specify additional options to configure the comparison behavior using a `Simulink.sdi.constraints.MatchesSignalOptions` object or using one or more name-value pair arguments on page 5-519. For more information about how tolerances and alignment options affect comparisons, see “How the Simulation Data Inspector Compares Data”.

**Note** The MatchesSignal constraint uses the Simulation Data Inspector to perform comparisons and requires a license for one of these products:

- Simulink
- Signal Processing Toolbox™
- MATLAB Coder
- Fixed-Point Designer

**Creation**

**Description**

`MatchesSignal(expVal)` creates a constraint that determines whether time series data representing an actual value matches the data specified by the expected value, `expVal` on page 5-0.
MatchesSignal(expVal,Name,Value) creates a constraint with additional properties on page 5-520 specified by one or more Name,Value pair arguments.

**Input Arguments**

**Name-Value Pair Arguments**

Specify optional comma-separated pairs of Name,Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside quotes. You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

Example: 'IgnoringDataTypes',true

**IgnoringDataTypes — Constraint behavior when data types do not match**  
false (default) | true

Configure constraint sensitivity to the actual value having a different data type from the expected value.

- **false** — The constraint checks whether the expected value and actual value data types match.
- **true** — The constraint is insensitive to data type mismatches between the expected value and actual value.

Example: 'IgnoringDataTypes',true

**IgnoringSignalsNotAligned — Constraint behavior when signals do not align**  
false (default) | true

Configure constraint sensitivity to one or more signals not aligning between the expected value and actual value.

- **false** — The constraint checks for signals that do not align between the expected value and actual value.
- **true** — The constraint is insensitive to one or more signals not aligning between the expected value and actual value.

Example: 'IgnoringSignalsNotAligned',true
IgnoringExtraData — Constraint behavior when the actual value covers a different time interval from the expected value
false (default) | true

Configure the constraint sensitivity to the actual value containing data outside of the time interval defined by the expected value.

- false — The constraint checks whether the actual value contains data outside the time interval defined by the expected value.
- true — The constraint is insensitive to the actual value containing data outside the time interval defined by the expected value.

Example: 'IgnoringExtraData',true

AligningBy — Signal alignment criteria
string array

Alignment criteria for the comparison. When the expected value contains more than one signal, the Simulation Data Inspector comparison algorithm aligns signals for comparison between the expected and actual values. By default, the Simulation Data Inspector aligns by data source, then by block path, then by SID, and then by signal name. For details about the Simulation Data Inspector alignment algorithm, see “How the Simulation Data Inspector Compares Data”.

Specify alignment criteria as a string array with one or more of these options listed in the order in which you want them considered:

- BlockPath — The path to the source block for the signal.
- SID — Simulink identifier. For more information, see “Locate Diagram Components Using Simulink Identifiers”.
- SignalName — The name of the signal.
- DataSource — Path of the variable.

Example: 'AligningBy',["SignalName","BlockPath"] specifies signal alignment by signal name and then by block path.

Properties

Expected — Expected value to use as the baseline for the comparison
time series data
Expected value to use as the baseline for the comparison. The data for the expected value must pair data values with time. The data can be in any format supported by the Simulation Data Inspector. The Simulation Data Inspector requires data in a format that associates sample values with time. Supported formats include timeseries, Structure with time, and Dataset.

Attributes:

GetAccess public
SetAccess immutable

MatchesSignalOptions — Constraint comparison configuration options
Simulink.sdi.constraints.MatchesSignalOptions

Simulink.sdi.constraints.MatchesSignalOptions object that specifies options to configure the comparison for the constraint. By default, the comparison uses the default alignment settings for the Simulation Data Inspector and has strict acceptance criteria, meaning that the comparison does not ignore any checks. You can use the MatchesSignalOptions property to configure how the constraint aligns signals in the expected and actual runs and whether tests using the constraint are sensitive to:

You can specify the MatchesSignalOptions property for the constraint using individual name-value pair arguments on page 5-519 or using the name-value pair 'WithOptions' with a Simulink.sdi.constraints.MatchesSignalOptions object as the value. When you specify the MatchesSignalOptions property using a MatchesSignalOptions object, you can use a single name-value pair to configure all the options for comparisons that use the constraint.

Attributes:

GetAccess public
SetAccess immutable

Data Types: Simulink.sdi.constraints.MatchesSignalOptions

AbsTol — Absolute tolerance for comparison
0 (default) | scalar

Absolute tolerance for comparisons using the constraint. For details about how the Simulation Data Inspector uses tolerance values in comparisons, see “How the Simulation Data Inspector Compares Data”.

5-521
Example: 0.1
Example: 2

Attributes:

GetAccess  public
SetAccess   immutable

Data Types: double

RelTol — Relative tolerance for comparison
0 (default) | scalar

Relative tolerance for comparisons using the constraint. For example, specify 0.1 to use a 10% tolerance for the comparison. For details about how the Simulation Data Inspector uses tolerance values in comparisons, see “How the Simulation Data Inspector Compares Data”.

Example: 0.05

Attributes:

GetAccess  public
SetAccess   immutable

Data Types: double

TimeTol — Time tolerance for comparison
0 (default) | scalar

Time tolerance for comparisons using the constraint, in seconds. For details about how the Simulation Data Inspector uses tolerance values in comparisons, see “How the Simulation Data Inspector Compares Data”.

Example: 0.001

Attributes:

GetAccess  public
SetAccess   immutable

Data Types: double
Examples

Create Tests for Time Series Comparisons

This example shows how to use the Simulink.sdi.constraints.MatchesSignal constraint to compare time series data using data generated in the workspace. You can use the MatchesSignal constraint to compare data from many sources, including measured test data in a file and logged simulation outputs.

When you use the MatchesSignal constraint, you can specify absolute, relative, and time tolerances to use in the comparison. The MatchesSignal constraint uses the Simulation Data Inspector comparison algorithm, which includes steps for alignment and synchronization. For more information about the algorithm, see “How the Simulation Data Inspector Compares Data”.

The MatchesSignal constraint uses the Simulation Data Inspector for the time series comparison and requires one of these products:

• Simulink®
• Signal Processing Toolbox™
• MATLAB Coder™
• Fixed-Point Designer™

Create Time Series Data

This example generates workspace data in timeseries format for illustrative purposes only. You can use the MatchesSignal constraint to compare data from other sources, such as simulation outputs or a file containing test data, and the constraint can compare time series data in any format supported by the Simulation Data Inspector.

In this example, the first signal is a timeseries object containing data for a sine wave with the name Wave Data.

time = 0:0.1:20;
sig1vals = sin(2*pi/5*time);
sig1_ts = timeseries(sig1vals,time);
sig1_ts.Name = 'Wave Data';

The second signal is also a sine wave at the same frequency, with a slight attenuation and sampled at a different rate. Because the comparison algorithm for the MatchesSignal
constraint includes a synchronization step, comparing these two signals with different 
time vectors does not always result in a test failure. The second signal is also named Wave 
Data so the alignment algorithm can pair the signals for comparison.

time2 = 0:0.05:20;
sig2vals = 0.98*sin(2*pi/5*time2);
sig2_ts = timeseries(sig2vals,time2);
sig2_ts.Name = 'Wave Data';

**Create a Test Case and Compare the Signals**

This example uses an interactive test for illustrative purposes. You can use the 
MatchesSignal constraint with other types of unit tests in the MATLAB testing 
framework. For example, to write class-based unit tests, see “Write Simple Test Case 
Using Classes” (MATLAB).

Create a TestCase instance for interactive use.

```matlab
import matlab.unittest.TestCase
testCase = TestCase.forInteractiveUse;

Compare sig1_ts and sig2_ts using the MatchesSignal constraint. Use sig1_ts for 
the expected value in the instance of the MatchesSignal constraint.

```matlab
import Simulink.sdi.constraints.MatchesSignal
testCase.verifyThat(sig2_ts,MatchesSignal(sig1_ts));
```

Verification failed.

---------------------
Framework Diagnostic:
---------------------
MatchesSignal failed.
---> The following aligned signals did not match:

<table>
<thead>
<tr>
<th>ActualValue</th>
<th>ExpectedValue</th>
</tr>
</thead>
<tbody>
<tr>
<td>{'&lt;Actual&gt;.Data'}</td>
<td>{'&lt;Expected&gt;.Data'}</td>
</tr>
</tbody>
</table>

---> Comparison snapshots captured to:
    --> C:\TEMP\Bdoc19b_1192687_6748\ibF7BE2B\11\Snapshot_980aa788-c4a3-4e85-b83e-0
The diagnostic information shows that the verification failed. Click the link to view the comparison snapshot of both the comparison signals and the difference signal. From the plot, you can see the attenuation creates the difference and causes the comparison to fail.

Specify an absolute tolerance of 0.02 for the instance of the `MatchesSignal` constraint to use in the comparison.

```matlab
testCase.verifyThat(sig2_ts, MatchesSignal(sig1_ts, 'AbsTol', 0.02));
```
Verification passed.

Configure MatchesSignal Constraint Behavior

This example shows how to configure the comparison behavior of a Simulink.sdi.constraints.MatchesSignal constraint using a Simulink.sdi.constraints.MatchesSignalOptions object. You can use a MatchesSignalOptions object to configure multiple comparison properties with a single name-value pair input in the construction of the MatchesSignal instance.

Create Time Series Data

This example creates workspace data to use for illustrative purposes only. You can use the MatchesSignal constraint to compare data from other sources, such as simulation outputs or a file containing test data, and the constraint can compare time series data in any format supported by the Simulation Data Inspector.

In this example, the first signal is a timeseries object with the name Wave Data that contains data for a sine wave.

```plaintext
time = 0:0.1:20;
sig1vals = sin(2*pi/5*time);
sig1_ts = timeseries(sig1vals,time);
sig1_ts.Name = 'Wave Data';
```

The second signal is also a sine wave at the same frequency, with a slight attenuation and sampled at a different rate. Because the comparison algorithm for the MatchesSignal constraint includes a synchronization step, comparing these two signals with different time vectors does not necessarily result in a test failure. The second signal is cast to single data type and also named Wave Data so the alignment algorithm can pair the signals for comparison.

```plaintext
time2 = 0:0.1:22;
sig2vals = single(0.98*sin(2*pi/5*time2));
sig2_ts = timeseries(sig2vals,time2);
sig2_ts.Name = 'Wave Data';
```

Configure a MatchesSignal Instance and Compare the Data

You can use a MatchesSignalOptions object to specify how an instance of the MatchesSignal constraint handles conditions such as mismatched data types and how signals are aligned between the actual and expected data sets. You can specify the properties of the MatchesSignalOptions object to suit the test you want to write. For
example, if you want your test to compare data of varying types and a data type match is not functionally relevant, you could set the **IgnoreDataTypes** property to **true**.

Create a **MatchesSignalOptions** object to create a **MatchesSignal** instance that ignores data type mismatches and data samples outside the time interval the actual and expected signals share.

```matlab
import Simulink.sdi.constraints.MatchesSignalOptions
opts = MatchesSignalOptions('IgnoringDataTypes',true,'IgnoringExtraData',true);
```

Create a **TestCase** instance for interactive use and compare `sig1_ts` and `sig2_ts` using the **MatchesSignal** constraint, the **MatchesSignalOptions** object, and an absolute tolerance of `0.02`.

```matlab
import matlab.unittest.TestCase
testCase = TestCase.forInteractiveUse;
import Simulink.sdi.constraints.MatchesSignal
testCase.verifyThat(sig2_ts,MatchesSignal(sig1_ts,'AbsTol',0.02,'WithOptions',opts))
```

Verification passed.

The test passes. When you run the same test again without the options defined by the **MatchesSignalOptions** object, the test fails due to the mismatched data types.

```matlab
testCase.verifyThat(sig2_ts,MatchesSignal(sig1_ts,'AbsTol',0.02))
```

Verification failed.
---------------------
Framework Diagnostic:
---------------------
**MatchesSignal** failed.
--> Datatype does not match for the following aligned signals:

<table>
<thead>
<tr>
<th>ActualValue</th>
<th>ActualDatatype</th>
<th>ExpectedValue</th>
<th>ExpectedDatatype</th>
</tr>
</thead>
<tbody>
<tr>
<td>{'&lt;Actual&gt;.Data'}</td>
<td>{'single'}</td>
<td>{'&lt;Expected&gt;.Data'}</td>
<td>{'double'}</td>
</tr>
</tbody>
</table>
---------------------

Stack Information:
---------------------
In C:\TEMP\Bdoc19b\1192687_6748\ibF7BE2B\11\tp42b18a65\simulink-ex71177165\ConfigureMatchesSignalConstraintBehaviorExample.mlx (ConfigureMatchesSignalConstraintBehaviorExample) at 16
In B:\matlab\toolbox\matlab\codetools\embeddedoutputs\+matlab\+internal\+editor\evaluateRegions.p (evaluateRegions) at 0
In B:\matlab\toolbox\matlab\codetools\embeddedoutputs\+matlab\+internal\+editor\EvaluationOutputsService.p (EvaluationOutputsService.evalRegions) at 0
In B:\matlab\toolbox\matlab\+matlab\+internal\+liveeditor\@LiveEditorUtilities\execute.m (execute) at 48
See Also
Simulink.sdi.constraints.MatchesSignalOptions |
matlab.unittest.TestCase | matlab.unittest.constraints Package |
matlab.unittest.constraints.Constraint | verifyThat

Topics
“How the Simulation Data Inspector Compares Data”
“Author Class-Based Unit Tests in MATLAB” (MATLAB)
“Write Script-Based Unit Tests” (MATLAB)
“Write Simple Test Case Using Classes” (MATLAB)
“Write Simple Test Case Using Functions” (MATLAB)

Introduced in R2019a
Simulink.sdi.constraints.MatchesSignalOptions class

Package: Simulink.sdi.constraints

Specify comparison options for Simulink.sdi.MatchesSignal constraint

Description

Use a Simulink.sdi.constraints.MatchesSignalOptions object to configure the comparison behavior and acceptance criteria for an instance of the Simulink.sdi.constraints.MatchesSignal constraint.

Note The MatchesSignal constraint uses the Simulation Data Inspector to perform comparisons and requires a license for one of these products:

- Simulink
- Signal Processing Toolbox
- MATLAB Coder
- Fixed-Point Designer

Creation

Description

MatchesSignalOptions creates an instance of the Simulink.sdi.constraints.MatchesSignalOptions class with properties that correspond to a constraint that does not ignore any checks and uses the default alignment settings for the Simulation Data Inspector.

MatchesSignalOptions(Name,Value) creates an instance of the Simulink.sdi.constraints.MatchesSignalOptions class with comparison
behavior using properties on page 5-530 specified by one or more Name, Value pair arguments.

Properties

**IgnoreDataTypes** — Constraint behavior when data types do not match
false (default) | true

Constraint sensitivity to the actual value having a different data type from the expected value.

You can specify the IgnoreDataTypes property in the constructor using the 'IgnoringDataTypes' name-value pair argument.

**Attributes:**

GetAccess public
SetAccess public

**IgnoreSignalsNotAligned** — Constraint behavior when signals do not align
false (default) | true

Constraint sensitivity to one or more signals not aligning between the expected value and actual value.

You can specify the IgnoreSignalsNotAligned property in the constructor using the 'IgnoringSignalsNotAligned' name-value pair argument.

**Attributes:**

GetAccess public
SetAccess public

**IgnoreExtraData** — Constraint behavior when the actual value covers a different time interval from the expected value
false (default) | true

Constraint sensitivity to the actual value containing data outside of the time interval defined by the expected value.
You can specify the IgnoreExtraData property in the constructor using the 'IgnoringExtraData' name-value pair argument.

**Attributes:**

GetAccess  
SetAccess  

**Alignment — Signal alignment criteria**

string array

Alignment criteria for the comparison. When the expected value contains more than one signal, the Simulation Data Inspector comparison algorithm aligns signals for comparison between the expected and actual values. By default, the Simulation Data Inspector aligns by data source, then by block path, then by SID, and then by signal name. For details about the Simulation Data Inspector alignment algorithm, see “How the Simulation Data Inspector Compares Data”.

Specify alignment criteria as a string array with one or more of these options listed in the order in which you want them considered:

You can specify the Alignment property in the constructor using the 'AligningBy' name-value pair argument.

**Examples**

**Configure MatchesSignal Constraint Behavior**

This example shows how to configure the comparison behavior of a Simulink.sdi.constraints.MatchesSignal constraint using a Simulink.sdi.constraints.MatchesSignalOptions object. You can use a MatchesSignalOptions object to configure multiple comparison properties with a single name-value pair input in the construction of the MatchesSignal instance.

**Create Time Series Data**

This example creates workspace data to use for illustrative purposes only. You can use the MatchesSignal constraint to compare data from other sources, such as simulation
outputs or a file containing test data, and the constraint can compare time series data in any format supported by the Simulation Data Inspector.

In this example, the first signal is a `timeseries` object with the name `Wave Data` that contains data for a sine wave.

```matlab
time = 0:0.1:20;
sig1vals = sin(2*pi/5*time);
sig1_ts = timeseries(sig1vals,time);
sig1_ts.Name = 'Wave Data';
```

The second signal is also a sine wave at the same frequency, with a slight attenuation and sampled at a different rate. Because the comparison algorithm for the `MatchesSignal` constraint includes a synchronization step, comparing these two signals with different time vectors does not necessarily result in a test failure. The second signal is cast to `single` data type and also named `Wave Data` so the alignment algorithm can pair the signals for comparison.

```matlab
time2 = 0:0.1:22;
sig2vals = single(0.98*sin(2*pi/5*time2));
sig2_ts = timeseries(sig2vals,time2);
sig2_ts.Name = 'Wave Data';
```

**Configure a MatchesSignal Instance and Compare the Data**

You can use a `MatchesSignalOptions` object to specify how an instance of the `MatchesSignal` constraint handles conditions such as mismatched data types and how signals are aligned between the actual and expected data sets. You can specify the properties of the `MatchesSignalOptions` object to suit the test you want to write. For example, if you want your test to compare data of varying types and a data type match is not functionally relevant, you could set the `IgnoreDataTypes` property to `true`.

Create a `MatchesSignalOptions` object to create a `MatchesSignal` instance that ignores data type mismatches and data samples outside the time interval the actual and expected signals share.

```matlab
import Simulink.sdi.constraints.MatchesSignalOptions
opts = MatchesSignalOptions('IgnoringDataTypes',true,'IgnoringExtraData',true);
```

Create a `TestCase` instance for interactive use and compare `sig1_ts` and `sig2_ts` using the `MatchesSignal` constraint, the `MatchesSignalOptions` object, and an absolute tolerance of `0.02`.  

```matlab
```
import matlab.unittest.TestCase
testCase = TestCase.forInteractiveUse;

import Simulink.sdi.constraints.MatchesSignal
testCase.verifyThat(sig2_ts, MatchesSignal(sig1_ts, 'AbsTol', 0.02, 'WithOptions', opts))

Verification passed.

The test passes. When you run the same test again without the options defined by the MatchesSignalOptions object, the test fails due to the mismatched data types.

testCase.verifyThat(sig2_ts, MatchesSignal(sig1_ts, 'AbsTol', 0.02))

Verification failed.

----------
Framework Diagnostic:
----------
MatchesSignal failed.
--> Datatype does not match for the following aligned signals:

<table>
<thead>
<tr>
<th>ActualValue</th>
<th>ActualDatatype</th>
<th>ExpectedValue</th>
<th>ExpectedDatatype</th>
</tr>
</thead>
<tbody>
<tr>
<td>{'&lt;Actual&gt;.Data'}</td>
<td>'single'</td>
<td>{'&lt;Expected&gt;.Data'}</td>
<td>'double'</td>
</tr>
</tbody>
</table>

----------
Stack Information:
----------
In C:\TEMP\Bdoc19b_1192687_6748\ibF7BE2B\11\tp42b18a65\simulink-ex71177165\ConfigureMatchesSignalConstraintBehaviorExample.mlx (ConfigureMatchesSignalConstraintBehaviorExample) at 16
In B:\matlab\toolbox\matlab\codetools\embeddedoutputs\+matlab\+internal\+editor\evaluateRegions.p (evaluateRegions) at 0
In B:\matlab\toolbox\matlab\codetools\embeddedoutputs\+matlab\+internal\+editor\EvaluationOutputsService.p (EvaluationOutputsService.evalRegions) at 0
In B:\matlab\toolbox\+matlab\+internal\+liveeditor\@LiveEditorUtils\execute.m (execute) at 48
In B:\matlab\tools\examples\exampletools\+mwtools\liveCodeToDocbook.m (doRun) at 246
In B:\matlab\tools\examples\exampletools\+mwtools\liveCodeToDocbook.m (doRunConvert) at 202
In B:\matlab\tools\build_using_matlab\BML.m (BML) at 13

See Also
Simulink.sdi.constraints.MatchesSignal | matlab.unittest.TestCase|
matlab.unittest.constraints Package|
matlab.unittest.constraints.Constraint | verifyThat
Topics
“How the Simulation Data Inspector Compares Data”
“Author Class-Based Unit Tests in MATLAB” (MATLAB)
“Write Script-Based Unit Tests” (MATLAB)
“Write Simple Test Case Using Classes” (MATLAB)
“Write Simple Test Case Using Functions” (MATLAB)

Introduced in R2019a
Simulink.sdi.CustomSnapshot class

Package: Simulink.sdi

Specify settings for a snapshot without opening or affecting the Simulation Data Inspector

Description

Use a Simulink.sdi.CustomSnapshot object to specify settings for a snapshot you want to create without opening the Simulation Data Inspector or affecting the open session. Creating a snapshot using a Simulink.sdi.CustomSnapshot object is the best option for fully scripted workflows. You can specify the snapshot dimensions in pixels, the subplot layout, and limits for the x- and y-axes. You can use the clearSignals and plotOnSubplot methods to plot signals you want to include in the snapshot. To capture the snapshot, you can pass the Simulink.sdi.CustomSnapshot object as the value for the settings name-value pair for the Simulink.sdi.snapshot function or use the snapshot method.

Construction


Properties

Width — Image width
600 (default) | scalar

Image width, in pixels.
Example: 750

Height — Image height
400 (default) | scalar

Image height, in pixels.
Example: 500

**Rows — Number of subplot rows**

1 (default) | scalar

Number of subplot rows, specified as a scalar between 1 and 8, inclusive. Use Rows and Columns to set your desired subplot layout.

Example: 2

**Columns — Number of subplot columns**

1 (default) | scalar

Number of subplot columns, specified as a scalar between 1 and 8, inclusive. Use Rows and Columns to set your desired subplot layout.

Example: 3

**TimeSpan — X-axis limits**

2x1 matrix

Limits for the time axis in the snapshot. The time axis limits are the same for all subplots. By default, the time axis adjusts to accommodate the largest time range of the plotted signals.

Example: [0 20]

**YRange — Y-axis limits**

Cell array

Cell array of 1-by-2 matrices specifying the y-axis limits for all subplots in the custom snapshot. By default, the snapshot operation auto-scales the y-axis to fit the plotted data.

Example: {
  [[-10 10], [0 100]]
}

**Methods**

clearSignals Clear signals plotted on subplots of a Simulink.sdi.CustomSnapshot object

plotOnSubPlot Plot signals on Simulink.sdi.CustomSnapshot object subplots

snapshot Create a custom snapshot
Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects (MATLAB).

Examples

Copy View Settings to a Run

This example shows how to copy view settings from one run to another and how to create figures using the Simulink.sdi.CustomSnapshot object.

Simulate Your Model and Get a Run Object

Configure the vdp model to save output data, and run a simulation to create data.

```matlab
load_system('vdp')
set_param('vdp','SaveFormat','Dataset','SaveOutput','on')
set_param('vdp/Mu','Gain','1');
sim('vdp');
```

Use the Simulation Data Inspector programmatic interface to access the run data.

```matlab
runIndex = Simulink.sdi.getRunCount;
runID = Simulink.sdi.getRunIDByIndex(runIndex);
vdpRun = Simulink.sdi.getRun(runID);
```

Modify Signal View Settings

Use the Simulink.sdi.Run object to access signals in the run. Then, modify the signal view settings. This example specifies the line color and style for each signal. The view settings for the run comprise the view settings for each signal and view settings specified for the plot area.

```matlab
sig1 = vdpRun.getSignalByIndex(1);
sig2 = vdpRun.getSignalByIndex(2);

sig1.LineColor = [0 0 1];
sig1.LineDashed = '-..';

sig2.LineColor = [1 0 0];
sig2.LineDashed = ':';
```
Capture a Snapshot from the Simulation Data Inspector

Create a Simulink.sdi.CustomSnapshot object and use the Simulink.sdi.snapshot function to programmatically capture a snapshot of the contents of the Simulation Data Inspector.

```matlab
snap = Simulink.sdi.CustomSnapshot;
```

You can use properties of the Simulink.sdi.CustomSnapshot object to configure the plot settings, like the subplot layout and axis limits, and to plot signals. When you use a Simulink.sdi.CustomSnapshot object to create your figure, these plot settings do not affect the Simulation Data Inspector.

```matlab
snap.Rows = 2;
snap.YRange = {[[-2.25 2.25],[-3 3]};
snap.plotOnSubPlot(1,1,sig1,true);
snap.plotOnSubPlot(2,1,sig2,true)
```

Use Simulink.sdi.snapshot to generate the figure you specified in the properties of the Simulink.sdi.CustomSnapshot object.

```matlab
fig = Simulink.sdi.snapshot("from","custom","to","figure","settings",snap);
```
Copy the View Settings to a New Simulation Run

Simulate the model again, with a different Mu value. Use the Simulation Data Inspector programmatic interface to access the simulation data.

```matlab
set_param('vdp/Mu','Gain','5')
sim('vdp');
runIndex2 = Simulink.sdi.getRunCount;
runID2 = Simulink.sdi.getRunIDByIndex(runIndex2);
run2 = Simulink.sdi.getRun(runID2);
```

To create a plot of the new output data that looks like the one you created in the previous step, you can copy the view settings to the run in a single line of code using

```matlab
Simulink.sdi.CustomSnapshot class
```
Simulink.sdi.copyRunViewSettings. The Simulink.sdi.copyRunViewSettings function does not automatically update plot settings in Simulink.sdi.CustomSnapshot objects, so specify the plot input as false.

\[
\text{sigIDs} = \text{Simulink.sdi.copyRunViewSettings}(\text{runID}, \text{runID2}, \text{false});
\]

**Capture a Snapshot of the New Simulation Run**

Use the Simulink.sdi.CustomSnapshot object to capture a snapshot of the new simulation run. First, clear the signals from the subplots. Then, plot the signals from the new run and capture another snapshot.

snap.clearSignals
snap.YRange = {[-2.25 2.25], [-8 8]};
snap.plotOnSubPlot(1,1,sigIDs(1),true)
snap.plotOnSubPlot(2,1,sigIDs(2),true)

fig = snap.snapshot("to","figure");
See Also
Simulink.sdi.Signal | Simulink.sdi.snapshot

Topics
“Inspect and Compare Data Programmatically”
“Create Plots Using the Simulation Data Inspector”

Introduced in R2018a
clearSignals

**Class:** Simulink.sdi.CustomSnapshot  
**Package:** Simulink.sdi

Clear signals plotted on subplots of a Simulink.sdi.CustomSnapshot object

**Syntax**

```
snap.clearSignals
```

**Description**

`snap.clearSignals` clears plotted signals from all subplots in the Simulink.sdi.CustomSnapshot object, `snap`. Using the `clearSignals` method does not affect any subplots or signals in your open Simulation Data Inspector session.

**Examples**

**Copy View Settings to a Run**

This example shows how to copy view settings from one run to another and how to create figures using the Simulink.sdi.CustomSnapshot object.

**Simulate Your Model and Get a Run Object**

Configure the `vdp` model to save output data, and run a simulation to create data.

```
load_system('vdp')
set_param('vdp','SaveFormat','Dataset','SaveOutput','on')
set_param('vdp/Mu','Gain','1');
sim('vdp');
```

Use the Simulation Data Inspector programmatic interface to access the run data.
runIndex = Simulink.sdi.getRunCount;
runID = Simulink.sdi.getRunIDByIndex(runIndex);
vdpRun = Simulink.sdi.getRun(runID);

**Modify Signal View Settings**

Use the Simulink.sdi.Run object to access signals in the run. Then, modify the signal view settings. This example specifies the line color and style for each signal. The view settings for the run comprise the view settings for each signal and view settings specified for the plot area.

sig1 = vdpRun.getSignalByIndex(1);
sig2 = vdpRun.getSignalByIndex(2);

sig1.LineColor = [0 0 1];
sig1.LineDashed = '-.';

sig2.LineColor = [1 0 0];
sig2.LineDashed = ':';

**Capture a Snapshot from the Simulation Data Inspector**

Create a Simulink.sdi.CustomSnapshot object and use the Simulink.sdi.snapshot function to programmatically capture a snapshot of the contents of the Simulation Data Inspector.

snap = Simulink.sdi.CustomSnapshot;

You can use properties of the Simulink.sdi.CustomSnapshot object to configure the plot settings, like the subplot layout and axis limits, and to plot signals. When you use a Simulink.sdi.CustomSnapshot object to create your figure, these plot settings do not affect the Simulation Data Inspector.

snap.Rows = 2;
snap.YRange = {{[-2.25 2.25],[-3 3]});
snap.plotOnSubPlot(1,1,sig1,true)
snap.plotOnSubPlot(2,1,sig2,true)

Use Simulink.sdi.snapshot to generate the figure you specified in the properties of the Simulink.sdi.CustomSnapshot object.

fig = Simulink.sdi.snapshot("from","custom","to","figure","settings",snap);
Copy the View Settings to a New Simulation Run

Simulate the model again, with a different \( \mu \) value. Use the Simulation Data Inspector programmatic interface to access the simulation data.

```matlab
set_param('vdp/Mu','Gain','5')
sim('vdp');
runIndex2 = Simulink.sdi.getRunCount;
runID2 = Simulink.sdi.getRunIDByIndex(runIndex2);
run2 = Simulink.sdi.getRun(runID2);
```

To create a plot of the new output data that looks like the one you created in the previous step, you can copy the view settings to the run in a single line of code using
Simulink.sdi.copyRunViewSettings. The Simulink.sdi.copyRunViewSettings function does not automatically update plot settings in Simulink.sdi.CustomSnapshot objects, so specify the plot input as false.

```matlab
sigIDs = Simulink.sdi.copyRunViewSettings(runID,runID2,false);
```

**Capture a Snapshot of the New Simulation Run**

Use the Simulink.sdi.CustomSnapshot object to capture a snapshot of the new simulation run. First, clear the signals from the subplots. Then, plot the signals from the new run and capture another snapshot.

```matlab
snap.clearSignals
snap.YRange = {{[-2.25 2.25]},{[-8 8]}};
snap.plotOnSubPlot(1,1,sigIDs(1),true)
snap.plotOnSubPlot(2,1,sigIDs(2),true)

fig = snap.snapshot("to","figure");
```
See Also
Simulink.sdi.CustomSnapshot | Simulink.sdi.clear |
Simulink.sdi.clearPreferences | Simulink.sdi.snapshot | plotOnSubPlot | snapshot

Topics
“Inspect and Compare Data Programmatically“
“Create Plots Using the Simulation Data Inspector“

Introduced in R2018a
plotOnSubPlot

Class: Simulink.sdi.CustomSnapshot
Package: Simulink.sdi

Plot signals on Simulink.sdi.CustomSnapshot object subplots

Syntax

snap.plotOnSubPlot(row, column, signal, plot)

Description

snap.plotOnSubPlot(row, column, signal, plot) plots the signal on the subplot in the Simulink.sdi.CustomSnapshot object, snap, specified by row and column when plot is true. When plot is false, plotOnSubPlot clears the signal from the subplot.

Input Arguments

row — Subplot row
class

Row for subplot on which you want to plot a signal. Specify row as a value from 1 through 8, inclusive.

Example: 2

column — Subplot column
class

Column for subplot on which you want to plot a signal. Specify column as a value from 1 through 8, inclusive.

Example: 3
**signal — Signal to plot**
Simulink.sdi.Signal | signal ID

Signal ID or Simulink.sdi.Signal object corresponding to the signal you want to plot.
Example: sigID

**plot — Plot indicator**
logical

Logical indicator of whether to plot or clear the signal from the subplot.
• true – Plot the signal.
• false – Clear the signal.
Example: true
Data Types: logical

**Examples**

**Copy View Settings to a Run**

This example shows how to copy view settings from one run to another and how to create figures using the Simulink.sdi.CustomSnapshot object.

**Simulate Your Model and Get a Run Object**

Configure the vdp model to save output data, and run a simulation to create data.

```matlab
load_system('vdp')
set_param('vdp','SaveFormat','Dataset','SaveOutput','on')
set_param('vdp/Mu','Gain','1');
sim('vdp');
```

Use the Simulation Data Inspector programmatic interface to access the run data.

```matlab
runIndex = Simulink.sdi.getRunCount;
runID = Simulink.sdi.getRunIDByIndex(runIndex); vdpRun = Simulink.sdi.getRun(runID);
```
Modify Signal View Settings

Use the `Simulink.sdi.Run` object to access signals in the run. Then, modify the signal view settings. This example specifies the line color and style for each signal. The view settings for the run comprise the view settings for each signal and view settings specified for the plot area.

```matlab
sig1 = vdpRun.getSignalByIndex(1);
sig2 = vdpRun.getSignalByIndex(2);

sig1.LineColor = [0 0 1];
sig1.LineDashed = '-.);

sig2.LineColor = [1 0 0];
sig2.LineDashed = ':;`
```

Capture a Snapshot from the Simulation Data Inspector

Create a `Simulink.sdi.CustomSnapshot` object and use the `Simulink.sdi.snapshot` function to programmatically capture a snapshot of the contents of the Simulation Data Inspector.

```matlab
snap = Simulink.sdi.CustomSnapshot;

You can use properties of the `Simulink.sdi.CustomSnapshot` object to configure the plot settings, like the subplot layout and axis limits, and to plot signals. When you use a `Simulink.sdi.CustomSnapshot` object to create your figure, these plot settings do not affect the Simulation Data Inspector.

```matlab
snap.Rows = 2;
snap.YRange = {[-2.25 2.25],[-3 3]};
snap.plotOnSubPlot(1,1,sig1,true)
snap.plotOnSubPlot(2,1,sig2,true)
```

Use `Simulink.sdi.snapshot` to generate the figure you specified in the properties of the `Simulink.sdi.CustomSnapshot` object.

```matlab
fig = Simulink.sdi.snapshot("from","custom","to","figure","settings",snap);
```
Copy the View Settings to a New Simulation Run

Simulate the model again, with a different Mu value. Use the Simulation Data Inspector programmatic interface to access the simulation data.

```matlab
set_param('vdp/Mu','Gain','5')
sim('vdp')
runIndex2 = Simulink.sdi.getRunCount;
runID2 = Simulink.sdi.getRunIDByIndex(runIndex2);
run2 = Simulink.sdi.getRun(runID2);
```

To create a plot of the new output data that looks like the one you created in the previous step, you can copy the view settings to the run in a single line of code using
Simulink.sdi.copyRunViewSettings. The Simulink.sdi.copyRunViewSettings function does not automatically update plot settings in Simulink.sdi.CustomSnapshot objects, so specify the plot input as false.

```
sigIDs = Simulink.sdi.copyRunViewSettings(runID, runID2, false);
```

**Capture a Snapshot of the New Simulation Run**

Use the Simulink.sdi.CustomSnapshot object to capture a snapshot of the new simulation run. First, clear the signals from the subplots. Then, plot the signals from the new run and capture another snapshot.

```
snap.clearSignals
snap.YRange = {{-2.25 2.25}, [-8 8]};
snap.plotOnSubPlot(1,1,sigIDs(1),true)
snap.plotOnSubPlot(2,1,sigIDs(2),true)

fig = snap.snapshot("to","figure");
```
See Also

Simulink.sdi.CustomSnapshot | Simulink.sdi.Signal |
Simulink.sdi.snapshot | clearSignals | snapshot

Topics

“Inspect and Compare Data Programmatically”
“Create Plots Using the Simulation Data Inspector”

Introduced in R2018a
snapshot

Class: Simulink.sdi.CustomSnapshot  
Package: Simulink.sdi

Create a custom snapshot

Syntax

fig = snap.snapshot  
[fig,image] = snap.snapshot  
fig = snap.snapshot(Name,Value)  
[fig,image] = snap.snapshot

Description

fig = snap.snapshot creates a figure according to the properties of the Simulink.sdi.CustomSnapshot object, snap, and returns the handle for the figure, fig.

[fig,image] = snap.snapshot creates a figure according to the properties of the Simulink.sdi.CustomSnapshot object, snap, and returns the handle for the figure, fig, and an array of image data, image.

fig = snap.snapshot(Name,Value) creates a figure according to the properties of the Simulink.sdi.CustomSnapshot object, snap, with additional options specified by one or more Name,Value pair arguments. This syntax returns the figure handle, fig.

[fig,image] = snap.snapshot creates a figure according to the properties of the Simulink.sdi.CustomSnapshot object, snap, with additional options specified by one or more Name,Value pair arguments. This syntax returns the figure handle, fig, and an array of image data, image.
Input Arguments

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name, Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside quotes. You can specify several name and value pair arguments in any order as Name1, Value1,..., NameN, ValueN.

Example: 'to', 'figure', 'props', {'Name', 'My Data'}

to — Type of snapshot to create
'image' (default) | 'figure' | 'file' | 'clipboard'

Type of snapshot to create.

• 'image' — Create a figure containing an image of the specified Simulation Data Inspector contents and return the figure handle and an array of image data. When you specify 'to', 'image', the fig and image outputs both have value.

• 'figure' — Replicate the specified Simulation Data Inspector contents in a MATLAB figure and return the figure handle. When you specify 'to', 'figure' the fig output has value, and the image output is empty.

• 'file' — Save a snapshot of the specified Simulation Data Inspector contents to a PNG file with the name specified by the filename name-value pair. If you do not specify a filename name-value pair, the file is named plots.png. When you specify 'to', 'file', the fig and image outputs are both empty.

• 'clipboard' — Copy the plots to your system clipboard. From the clipboard, you can paste the image into another program such as Microsoft Word. When you specify 'to', 'clipboard', the fig and image outputs are both empty.

Example: 'to', 'file'

Data Types: char | string

filename — Name for image file
'veplots.png' (default) | character array | string

Name of the image file to store the snapshot when you specify 'to', 'file'.

Example: 'filename', 'MyImage.png'

Data Types: char | string
**props — Properties to customize the figure**

Cell array

Figure properties, specified as a cell array. You can include settings for the figure properties described in Figure Properties.

Example: 'props',{ 'Name','MyData','NumberTitle','off'}

Data Types: char | string

**Output Arguments**

**fig — Figure handle**

Figure handle

Handle for the figure. When a figure is not created with your specified options, the fig output is empty.

**image — Image data**

Array

Array of image data. The image output has value when you use Simulink.sdi.snapshot without any input arguments or without a to name-value pair and when you specify 'to','image'.

**Examples**

**Copy View Settings to a Run**

This example shows how to copy view settings from one run to another and how to create figures using the Simulink.sdi.CustomSnapshot object.

**Simulate Your Model and Get a Run Object**

Configure the vdp model to save output data, and run a simulation to create data.

```matlab
load_system('vdp')
set_param('vdp','SaveFormat','Dataset','SaveOutput','on')
set_param('vdp/Mu','Gain','1');
sim('vdp');
```
Use the Simulation Data Inspector programmatic interface to access the run data.

```matlab
t = Simulink.sdi.getRunCount;
d = Simulink.sdi.getRunIDByIndex(t);
r = Simulink.sdi.getRun(d);
```

**Modify Signal View Settings**

Use the `Simulink.sdi.Run` object to access signals in the run. Then, modify the signal view settings. This example specifies the line color and style for each signal. The view settings for the run comprise the view settings for each signal and view settings specified for the plot area.

```matlab
d1 = r.getSignalByIndex(1);
d2 = r.getSignalByIndex(2);

d1.LineColor = [0 0 1];
d1.LineDashed = '-.';

d2.LineColor = [1 0 0];
d2.LineDashed = ':';
```

**Capture a Snapshot from the Simulation Data Inspector**

Create a `Simulink.sdi.CustomSnapshot` object and use the `Simulink.sdi.snapshot` function to programmatically capture a snapshot of the contents of the Simulation Data Inspector.

```matlab
sn = Simulink.sdi.CustomSnapshot;

You can use properties of the `Simulink.sdi.CustomSnapshot` object to configure the plot settings, like the subplot layout and axis limits, and to plot signals. When you use a `Simulink.sdi.CustomSnapshot` object to create your figure, these plot settings do not affect the Simulation Data Inspector.

```
sn.Rows = 2;
sn.YRange = {{[-2.25 2.25], [-3 3]}};
sn.plotOnSubPlot(1,1,d1,true)
sn.plotOnSubPlot(2,1,d2,true)
```

Use `Simulink.sdi.snapshot` to generate the figure you specified in the properties of the `Simulink.sdi.CustomSnapshot` object.

```matlab
fig = Simulink.sdi.snapshot("from","custom","to","figure","settings",sn);
```
Copy the View Settings to a New Simulation Run

Simulate the model again, with a different Mu value. Use the Simulation Data Inspector programmatic interface to access the simulation data.

```matlab
set_param('vdp/Mu','Gain','5')
sim('vdp');
```

```matlab
runIndex2 = Simulink.sdi.getRunCount; runID2 = Simulink.sdi.getRunIDByIndex(runIndex2); run2 = Simulink.sdi.getRun(runID2);
```

To create a plot of the new output data that looks like the one you created in the previous step, you can copy the view settings to the run in a single line of code using
Simulink.sdi.copyRunViewSettings. The Simulink.sdi.copyRunViewSettings function does not automatically update plot settings in Simulink.sdi.CustomSnapshot objects, so specify the plot input as false.

```matlab
sigIDs = Simulink.sdi.copyRunViewSettings(runID, runID2, false);
```

**Capture a Snapshot of the New Simulation Run**

Use the Simulink.sdi.CustomSnapshot object to capture a snapshot of the new simulation run. First, clear the signals from the subplots. Then, plot the signals from the new run and capture another snapshot.

```matlab
snap.clearSignals
snap.YRange = {{-2.25 2.25},[-8 8]};
snap.plotOnSubPlot(1,1,sigIDs(1),true)
snap.plotOnSubPlot(2,1,sigIDs(2),true)

fig = snap.snapshot("to","figure");
```
See Also
Figure Properties | Simulink.sdi.CustomSnapshot | Simulink.sdi.snapshot | clearSignals | plotOnSubPlot

Topics
“Inspect and Compare Data Programmatically”
“Create Plots Using the Simulation Data Inspector”

Introduced in R2018a
Simulink.sdi.DatasetRef class

Package: Simulink.sdi

Access data in Simulation Data Inspector repository

Description

The Simulink.sdi.DatasetRef class provides access to data in the Simulation Data Inspector repository without loading the entire set of data into memory. The class is compatible with the Simulink.SimulationData.DatasetRef class.

Construction

dsr_array = Simulink.sdi.DatasetRef constructs an array containing a Simulink.sdi.DatasetRef object for each run in the Simulation Data Inspector.

dsr_array = Simulink.sdi.DatasetRef(domain) creates an array containing a Simulink.sdi.DatasetRef object for each run, with the contents of each run limited to the selected domain.

dsr = Simulink.sdi.DatasetRef(runID) creates a Simulink.sdi.DatasetRef object of the run corresponding to the run identifier, runID.

dsr = Simulink.sdi.DatasetRef(runID, domain) creates a Simulink.sdi.DatasetRef object of the run corresponding to runID with the contents specified by domain.

dsr = Simulink.sdi.DatasetRef(runID, domain, repositoryPath) creates a Simulink.sdi.DatasetRef object of the run corresponding to runID including the contents specified by domain from a repository path specified by repositoryPath.

Input Arguments

domain — Specify contents of Simulink.sdi.DatasetRef objects
'signals' | 'outports' | []
Limits the contents included in the `Simulink.sdi.DatasetRef` object.

- `'signals'` includes only logged signals.
- `'outports'` includes only logged outports.
- `[]` includes all run data.

**runID — Run identifier**

integer

Specifies the run containing the data for the `Simulink.sdi.DatasetRef` object.

**repositoryPath — Path containing the run**

string | character vector

Specifies the location of the run containing the data for the `Simulink.sdi.DatasetRef` object.

## Properties

**Name — Run name**

character vector

The name of the run that corresponds with the `Simulink.sdi.DatasetRef` object.

Example: `'Run 1'`

**Run — Simulink.sdi.Run object**

`Simulink.sdi.Run` object

`Simulink.sdi.Run` object associated with the `Simulink.sdi.DatasetRef` object.

**numElements — Number of top-level elements in run**

`Simulink.sdi.Run` object

Number of top-level elements in the `Simulink.sdi.Run` object associated with the `Simulink.sdi.DatasetRef` object.
Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>compare</code></td>
<td>Compare runs with DatasetRef objects</td>
</tr>
<tr>
<td><code>getAsDatastore</code></td>
<td>Retrieve element as sdidatastore object</td>
</tr>
<tr>
<td><code>getElement</code></td>
<td>Retrieve DatasetRef element by index</td>
</tr>
<tr>
<td><code>getElementNames</code></td>
<td>Get character vectors of element names</td>
</tr>
<tr>
<td><code>getSignal</code></td>
<td>Return Signal object</td>
</tr>
<tr>
<td><code>plot</code></td>
<td>Open the Simulation Data Inspector to view and compare data</td>
</tr>
</tbody>
</table>

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects (MATLAB).

Examples

Compare Runs with the Simulink.sdi.DatasetRef Object

This example shows how to work with the Simulink.sdi.DatasetRef object by comparing two runs of the `ex_sldemo_absbrake` system with different desired slip ratios.

```matlab
% Simulate model ex_sldemo_absbrake to create a run of logged signals
load_system('ex_sldemo_absbrake')
sim('ex_sldemo_absbrake')

% Get the runID
runIDs = Simulink.sdi.getAllRunIDs;
runID = runIDs(end);

% Get the run object
brakeRun = Simulink.sdi.getRun(runID);

% Make a Simulink.sdi.DatasetRef object
run_DSRef = brakeRun.getDatasetRef;
```
% Get the names of the elements in the object
names = run_DSRef.getElementNames

names = 2x1 cell array
    {'yout'}
    {'slp' }

% Get yout bus
[yout, name, index] = run_DSRef.getElement(1);

% View signals in outputs
outputs = yout.Values

outputs = struct with fields:
    Ww: [1x1 timeseries]
    Vs: [1x1 timeseries]
    Sd: [1x1 timeseries]

% Get slp signal
slp = run_DSRef.getSignal('slp');

% Plot signal
slp.Checked = 'true';

% Create another run for a different Desired relative slip
set_param('ex_sldemo_absbrake/Desired relative slip', 'Value', '0.25')
sim('ex_sldemo_absbrake')

DSR_Runs = Simulink.sdi.DatasetRef;

% Compare the results from the two runs
[matches, mismatches, diffResult] = run_DSRef.compare(DSR_Runs(2));

% Open the Simulation Data Inspector to view signals
run_DSRef.plot

See Also
Simulink.SimulationData.DatasetRef | Simulink.sdi.Run | Simulink.sdi.WorkerRun | Simulink.sdi.WorkerRun.getDatasetRef
Topics
“Inspect and Compare Data Programatically”

Introduced in R2017b
compare

Class: Simulink.sdi.DatasetRef
Package: Simulink.sdi

Compare runs with DatasetRef objects

Syntax

[matches, mismatches, results] = dsrObj.compare(other)

Description

[matches, mismatches, results] = dsrObj.compare(other) returns the number of matches, number of mismatches, and comparison results for a comparison of data in a Simulink.sdi.DatasetRef object. The comparison results are returned as a Simulink.sdi.DiffRunResult object.

Input Arguments

other — Comparison data
MAT-file | variable

Comparison data, which can come from another Simulink.sdi.DatasetRef object, a Dataset in the workspace, or a MAT-file.

Example: 'data.mat'
Example: var

Output Arguments

matches — Number of matching signals
integer
Number of signals that matched within tolerance in the comparison.

\textit{mismatches} — \textit{Number of mismatched signals}
integer

Number of signals that did not match within tolerance in the comparison.

\textit{results} — \textit{Simulink.sdi.DiffRunResult object with comparison results}
Simulink.sdi.DiffRunResult object

Results of the comparison, returned in a Simulink.sdi.DiffRunResult object.

\section*{Examples}

\textbf{Compare Runs with the Simulink.sdi.DatasetRef Object}

This example shows how to work with the Simulink.sdi.DatasetRef object by comparing two runs of the ex_sldemo_absbrake system with different desired slip ratios.

% Simulate model ex_sldemo_absbrake to create a run of logged signals
load_system('ex_sldemo_absbrake')
sim('ex_sldemo_absbrake')

% Get the runID
runIDs = Simulink.sdi.getAllRunIDs;
runID = runIDs(end);

% Get the run object
brakeRun = Simulink.sdi.getRun(runID);

% Make a Simulink.sdi.DatasetRef object
run_DSRef = brakeRun.getDatasetRef;

% Get the names of the elements in the object
names = run_DSRef.getElementNames

names = 2x1 cell array
  {'}yout'}
  {'slp'}

% Get yout bus
[yout, name, index] = run_DSRef.getElement(1);
% View signals in outputs
outputs = yout.Values

outputs = struct with fields:
  Ww: [1x1 timeseries]
  Vs: [1x1 timeseries]
  Sd: [1x1 timeseries]

% Get slp signal
slp = run_DSRef.getSignal('slp');

% Plot signal
slp.Checked = 'true';

% Create another run for a different Desired relative slip
set_param('ex_sldemo_absbrake/Desired relative slip', 'Value', '0.25')
sim('ex_sldemo_absbrake')
DSR_Runs = Simulink.sdi.DatasetRef;

% Compare the results from the two runs
[matches, mismatches, diffResult] = run_DSRef.compare(DSR_Runs(2));

% Open the Simulation Data Inspector to view signals
run_DSRef.plot

Alternatives

Using the Simulation Data Inspector API, you could create runs for the data you want to compare and use Simulink.sdi.compareRuns for the comparison. You can also view runs created from simulation, import data to runs, and compare runs with the Simulation Data Inspector UI.

See Also
Simulink.sdi.DatasetRef | Simulink.sdi.DiffRunResult | Simulink.sdi.Run
| Simulink.sdi.compareRuns | Simulink.sdi.compareSignals |
Simulink.sdi.view
Topics
“Inspect and Compare Data Programmatically”
“How the Simulation Data Inspector Compares Data”

Introduced in R2017b
getAsDatastore

Class: Simulink.sdi.DatasetRef
Package: Simulink.sdi

Retrieve element as sdidatastore object

Syntax

[elementDatastore, name, index] = SDIDatasetRef.getAsDatastore(arg)

Description

[elementDatastore, name, index] = SDIDatasetRef.getAsDatastore(arg)
returns the requested element as a matlab.io.datastore.sdidatastoreobject,
along with the element name and index.

Input Arguments

arg — Element selection criterion
integer | character vector

Search criterion used to retrieve the element from the Simulink.sdi.DatasetRefobject. For name-based searches, specify arg as a character vector. For index-based searches, arg is an integer, representing the index of the desired element.

Example: 'MySignal'
Example: 3

Output Arguments

elementDatastore — Element as sdidatastore
sdidatastore object
Element as `matlab.io.datastore.sdidatastore` object.

**name — Element name**

character vector

The name of the element.

**index — Element index in DatasetRef object**

integer

The index of the element in the `Simulink.sdi.DatasetRef` object.

## Examples

### Create an sdidatastore Object for a Signal

This example shows how to create a `sdidatastore` object for a signal in a `Simulink.sdi.DatasetRef` object.

```matlab
% Simulate model sldemo_fuelsys to create a run of logged signals
sim('sldemo_fuelsys')

% Get the runID
runIDs = Simulink.sdi.getAllRunIDs;
runID = runIDs(end);

% Get the run object
fuelRun = Simulink.sdi.getRun(runID);

% Make a Simulink.sdi.DatasetRef object
run_DSRef = fuelRun.getDatasetRef;

% Get the names of the elements in the object
names = run_DSRef.getElementNames

names = 16x1 cell array
    {'CheckRange'                     }
    {'air_fuel_ratio'                 }
    {'speed'                          }
    {'map'                            }
    {'ego'                            }
    {'throttle'                        }
```
{'fuel'                         }
{'EGO Fault Switch:1'        }
{'Engine Speed Fault Switch:1'}
{'MAP Fault Switch:1'        }
{'Throttle Angle Fault Switch:1'}
{'ego_sw'                       }
{'engine_speed'                }
{'speed_sw'                     }
{'map_sw'                       }
{'throttle_sw'                  }

% Get sdidatastore object for fuel signal
fuel_ds = run_DSRef.getAsDatastore(10);

Alternatives

You can construct a sdidatastore object for a specified signal using
matlab.io.datastore.sdidatastore.

See Also

Simulink.sdi.DatasetRef|Simulink.sdi.DatasetRef.getElement|
matlab.io.datastore.SimulationDatastore|
matlab.io.datastore.sdidatastore

Topics

“Inspect and Compare Data Programmatically”

Introduced in R2017b
**getElement**

*Class:* Simulink.sdi.DatasetRef  
*Package:* Simulink.sdi

Retrieve DatasetRef element by index

**Syntax**

[element, name, index] = SDIdatasetRef.getElement(index)

**Description**

[element, name, index] = SDIdatasetRef.getElement(index) returns the element within the Run in the Simulink.sdi.DatasetRef object at the specified index.

**Input Arguments**

*index — Index of element*

integer

Location of the element in the Simulink.sdi.DatasetRef object.

**Output Arguments**

*element — Run element in the DatasetRef object*

signal

Element from the run in the Simulink.sdi.DatasetRef object.

*name — Element name*

character vector

Name of the element retrieved from the Simulink.sdi.DatasetRef object.
**index — Location of element**

integer

Location of the element within the Simulink.sdi.DatasetRef object.

**Examples**

**Compare Runs with the Simulink.sdi.DatasetRef Object**

This example shows how to work with the Simulink.sdi.DatasetRef object by comparing two runs of the ex_sldemo_absbrake system with different desired slip ratios.

```matlab
% Simulate model ex_sldemo_absbrake to create a run of logged signals
load_system('ex_sldemo_absbrake')
sim('ex_sldemo_absbrake')

% Get the runID
runIDs = Simulink.sdi.getAllRunIDs;
runID = runIDs(end);

% Get the run object
brakeRun = Simulink.sdi.getRun(runID);

% Make a Simulink.sdi.DatasetRef object
run_DSRef = brakeRun.getDatasetRef;

% Get the names of the elements in the object
names = run_DSRef.getElementNames

names = 2x1 cell array
    {'yout'}
    {'slp' }

% Get yout bus
[yout, name, index] = run_DSRef.getElement(1);

% View signals in outputs
outputs = yout.Values

outputs = struct with fields:
    Ww: [1x1 timeseries]```
Vs: [1x1 timeseries]
Sd: [1x1 timeseries]

% Get slp signal
slp = run_DSRef.getSignal('slp');

% Plot signal
slp.Checked = 'true';

% Create another run for a different Desired relative slip
set_param('ex_sldemo_absbrake/Desired relative slip', 'Value', '0.25')
sim('ex_sldemo_absbrake')
DSR_Runs = Simulink.sdi.DatasetRef;

% Compare the results from the two runs
[matches, mismatches, diffResult] = run_DSRef.compare(DSR_Runs(2));

% Open the Simulation Data Inspector to view signals
run_DSRef.plot

See Also
Simulink.sdi.DatasetRef | Simulink.sdi.DatasetRef.getElementNames | Simulink.sdi.DatasetRef.getSignal

Topics
“Inspect and Compare Data Programmatically”

Introduced in R2017b
getElementNames

Class: Simulink.sdi.DatasetRef
Package: Simulink.sdi

Get character vectors of element names

Syntax

names = dsrObj.getElementNames

Description

names = dsrObj.getElementNames returns a cell array of character vectors containing the names of the elements in dsrObj.

Output Arguments

names — Element names
cell array

Names of the top level elements in the Simulink.sdi.DatasetRef object in a cell array.

Examples

Compare Runs with the Simulink.sdi.DatasetRef Object

This example shows how to work with the Simulink.sdi.DatasetRef object by comparing two runs of the ex_sldemo_absbrake system with different desired slip ratios.

% Simulate model ex_sldemo_absbrake to create a run of logged signals
load_system('ex_sldemo_absbrake')
sim('ex_sldemo_absbrake')

% Get the runID
runIDs = Simulink.sdi.getAllRunIDs;
runID = runIDs(end);

% Get the run object
brakeRun = Simulink.sdi.getRun(runID);

% Make a Simulink.sdi.DatasetRef object
run_DSRef = brakeRun.getDatasetRef;

% Get the names of the elements in the object
names = run_DSRef.getElementNames
names = 2x1 cell array
    {'yout'}
    {'slp'}

% Get yout bus
[yout, name, index] = run_DSRef.getElement(1);

% View signals in outputs
outputs = yout.Values
outputs = struct with fields:
    Ww: [1x1 timeseries]
    Vs: [1x1 timeseries]
    Sd: [1x1 timeseries]

% Get slp signal
slp = run_DSRef.getSignal('slp');

% Plot signal
slp.Checked = 'true';

% Create another run for a different Desired relative slip
set_param('ex_sldemo_absbrake/Desired relative slip', 'Value', '0.25')
sim('ex_sldemo_absbrake')
DSR_Runs = Simulink.sdi.DatasetRef;

% Compare the results from the two runs
[matches, mismatches, diffResult] = run_DSRef.compare(DSR_Runs(2));
% Open the Simulation Data Inspector to view signals
run_DSRef.plot

See Also
Simulink.SimulationData.DatasetRef | Simulink.sdi.DatasetRef |
Simulink.sdi.DatasetRef.getElement | Simulink.sdi.DatasetRef.getSignal

Topics
“Inspect and Compare Data Programmatically”

Introduced in R2017b
getSignal

Class: Simulink.sdi.DatasetRef
Package: Simulink.sdi

Return Signal object

Syntax

sigObj = SDIDatasetRef.getSignal(searchArg)

Description

sigObj = SDIDatasetRef.getSignal(searchArg) returns the Simulink.sdi.Signal object corresponding to the search argument, searchArg.

Input Arguments

searchArg — Search parameter
character vector | integer

The search parameters to select the Simulink.sdi.Signal object. The searchArg can be a character vector or string targeting a signal name or an integer for an index-based search.

Example: 'throttle'
Example: 2

Output Arguments

sigObj — Simulink.sdi.Signal object
Simulink.sdi.Signal object

The Simulink.sdi.Signal object corresponding to the search query.
Examples

Compare Runs with the Simulink.sdi.DatasetRef Object

This example shows how to work with the Simulink.sdi.DatasetRef object by comparing two runs of the `ex_sldemo_absbrake` system with different desired slip ratios.

```matlab
% Simulate model ex_sldemo_absbrake to create a run of logged signals
load_system('ex_sldemo_absbrake')
sim('ex_sldemo_absbrake')

% Get the runID
runIDs = Simulink.sdi.getAllRunIDs;
runID = runIDs(end);

% Get the run object
brakeRun = Simulink.sdi.getRun(runID);

% Make a Simulink.sdi.DatasetRef object
run_DSRef = brakeRun.getDatasetRef;

% Get the names of the elements in the object
names = run_DSRef.getElementNames;

names = 2x1 cell array
  {'yout'}
  {'slp' }

% Get yout bus
[yout, name, index] = run_DSRef.getElement(1);

% View signals in outputs
outputs = yout.Values

outputs = struct with fields:
  Ww: [1x1 timeseries]
  Vs: [1x1 timeseries]
  Sd: [1x1 timeseries]

% Get slp signal
slp = run_DSRef.getSignal('slp');

% Plot signal
slp.Checked = 'true';

% Create another run for a different Desired relative slip
set_param('ex_sldemo_absbrake/Desired relative slip', 'Value', '0.25')
sim('ex_sldemo_absbrake')
DSR_Runs = Simulink.sdi.DatasetRef;

% Compare the results from the two runs
[matches, mismatches, diffResult] = run_DSRef.compare(DSR_Runs(2));

% Open the Simulation Data Inspector to view signals
run_DSRef.plot

Alternatives

If the signal is a top-level element in the Simulink.sdi.DatasetRef object, you can use the Simulink.sdi.DatasetRef.getElement method to get the Simulink.sdi.Signal object by index.

See Also

Simulink.sdi.DatasetRef | Simulink.sdi.DatasetRef.getElement | Simulink.sdi.getSignal

Topics

“Inspect and Compare Data Programmatically”

Introduced in R2017b
plot

Class: Simulink.sdi.DatasetRef
Package: Simulink.sdi

Open the Simulation Data Inspector to view and compare data

Syntax

SDIDatasetRef.plot

Description

SDIDatasetRef.plot opens the Simulation Data Inspector, where you can view and compare runs and signals.

Examples

Compare Runs with the Simulink.sdi.DatasetRef Object

This example shows how to work with the Simulink.sdi.DatasetRef object by comparing two runs of the ex_sldemo_absbrake system with different desired slip ratios.

% Simulate model ex_sldemo_absbrake to create a run of logged signals
load_system('ex_sldemo_absbrake')
sim('ex_sldemo_absbrake')

% Get the runID
runIDs = Simulink.sdi.getAllRunIDs;
runID = runIDs(end);

% Get the run object
brakeRun = Simulink.sdi.getRun(runID);

% Make a Simulink.sdi.DatasetRef object
run_DSRef = brakeRun.getDatasetRef;
% Get the names of the elements in the object
names = run_DSRef.getElementNames

names = 2x1 cell array
    {'yout'}
    {'slp' }

% Get yout bus
[yout, name, index] = run_DSRef.getElement(1);

% View signals in outputs
outputs = yout.Values

outputs = struct with fields:
    WW: [1x1 timeseries]
    Vs: [1x1 timeseries]
    Sd: [1x1 timeseries]

% Get slp signal
slp = run_DSRef.getSignal('slp');

% Plot signal
slp.Checked = 'true';

% Create another run for a different Desired relative slip
set_param('ex_sldemo_absbrake/Desired relative slip', 'Value', '0.25')
sim('ex_sldemo_absbrake')
DSR_Runs = Simulink.sdi.DatasetRef;

% Compare the results from the two runs
[matches, mismatches, diffResult] = run_DSRef.compare(DSR_Runs(2));

% Open the Simulation Data Inspector to view signals
run_DSRef.plot
Alternatives

You can use the Simulink.sdi.view function to open the Simulation Data Inspector. For information on using the UI to open the Simulation Data Inspector, see “View Data with the Simulation Data Inspector”.

See Also

Simulink.sdi.DatasetRef | Simulink.sdi.Signal.plotOnSubPlot | Simulink.sdi.setSubPlotLayout | Simulink.sdi.view

Topics

“Inspect and Compare Data Programmatically”

Introduced in R2017b
Simulink.sdi.DiffRunResult class

Package: Simulink.sdi

Access run comparison metadata

Description

The Simulink.sdi.DiffRunResult class provides access to the run comparison metadata. You can use the getSignalByIndex method to access the data and comparison results for each signal in the run comparison.

Construction

DiffRunResultObj = Simulink.sdi.compareRuns(runID1, runID2) returns a Simulink.sdi.DiffRunResult object to provide access to the comparison results from comparing the runs corresponding to runID1 and runID2.

Input Arguments

runID1 — Baseline run identifier
integer

Numeric run identifier for the Baseline run in the comparison.

runID2 — Compare to run identifier
integer

Numeric identifier for the Compare to run in the comparison.

Properties

MatlabVersion — Version used
character vector

Version of MATLAB used.
RunID1 — Baseline signal run ID
integer

Run identifier for the Baseline signal of the comparison.

RunID2 — Compare to signal run ID
integer

Run identifier for the Compare to signal of the comparison.

Count — Number of signals compared
integer

Number of signals aligned between the two runs in the comparison. For more information on how signals are aligned for comparisons, see “How the Simulation Data Inspector Compares Data”.

DateCreated — Object creation date
datetime

Date and time the Simulink.sdi.DiffRunResult object was created.

Data Types: datetime

GlobalTolerance — Global tolerance values used for the run comparison
structure

Global tolerance values used for the run comparison. The GlobalTolerance property value is a structure with fields:

- AbsTol — Global absolute tolerance used for the run comparison.
- RelTol — Global relative tolerance used for the run comparison.
- TimeTol — Global time tolerance used for the run comparison.

For more information about how tolerances are used and calculated for comparisons, see “How the Simulation Data Inspector Compares Data”.

Summary — Comparison results summary
structure

Comparison results summary. The summary is a structure with a field for each possible Status a comparison result signal could have. The value of each field indicates the number of signals in the run comparison with the corresponding Status.
• **WithinTolerance** — Signal comparison completed, and all data points compared fell within the specified tolerance.
• **OutOfRange** — Signal comparison completed, and some data points compared fell outside of the specified tolerance.
• **Unaligned** — Signal from the **Baseline** run did not align with a signal in the **Compare to** run.
• **Empty** — Aligned **Baseline** or **Compare to** signal contains no data.
• **EmptySynced** — Synchronized **Baseline** or **Compare to** signal contains no data.
• **Canceled** — Signal result not computed because the user canceled the comparison.
• **Pending** — Comparison is in progress and the signal result computation has not started.
• **Processing** — Signal result computation in progress.
• **UnitsMismatch** — **Baseline** and **Compare to** signal units do not match.
• **Unsupported** — Simulation Data Inspector comparison algorithm does not support this type of signal. For example, signals with data types that lose precision when converted to **double** are not supported.

For more information about alignment, tolerances, and synchronization, see “How the Simulation Data Inspector Compares Data”.

### Methods

getResultByIndex  
Return signal comparison result

### Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects (MATLAB).

### Examples

**Analyze Simulation Data with Signal Tolerances**

You can change tolerance values on a signal-by-signal basis to evaluate the effect of a model parameter change. This example uses the **slexAircraftExample** model and the
Simulation Data Inspector to evaluate the effect of changing the time constant for the low-pass filter following the control input.

**Setup**

Load the model, and mark the $q$, rad/sec and $\alpha$, rad signals for logging. Then, simulate the model to create the baseline run.

```matlab
% Load example model
load_system('slexAircraftExample')

% Mark the q, rad/sec and alpha, rad signals for logging
Simulink.sdi.markSignalForStreaming('slexAircraftExample/Aircraft Dynamics Model',3,'on')
Simulink.sdi.markSignalForStreaming('slexAircraftExample/Aircraft Dynamics Model',4,'on')

% Simulate system
out_1 = sim('slexAircraftExample');
```

**Modify Model Parameter**

Modify the model parameter $Ts$ in the model workspace to change the time constant of the input low-pass filter.

```matlab
% Change input filter time constant
modelWorkspace = get_param('slexAircraftExample','modelworkspace');
assignin(modelWorkspace,'Ts',1)

% Simulate again
out_2 = sim('slexAircraftExample');
```

**Compare Runs and Inspect Results**

Use the `Simulink.sdi.compareRuns` function to compare the data from the simulations. Then, inspect the match property of the signal result to see whether the signals fell within the default tolerance of 0.

```matlab
% Get run data
runIDs = Simulink.sdi.getAllRunIDs;
runID1 = runIDs(end - 1);
runID2 = runIDs(end);

% Compare runs
diffRun1 = Simulink.sdi.compareRuns(runID1,runID2);

% Get signal result
sig1Result1 = getResultByIndex(diffRun1,1);
sig2Result1 = getResultByIndex(diffRun1,2);

% Check whether signals matched
sig1Result1.Status
ans =
OutOfTolerance

sig2Result1.Status
ans =
OutOfTolerance

**Compare Runs with Signal Tolerances**

The signals did not match within the default tolerance of 0. To further analyze the effect of the time constant change, add signal tolerances to the comparison with the baseline signal properties to determine the tolerance required for a pass. This example uses a combination of time and absolute tolerances.

% Get signal object for sigID1
run1 = Simulink.sdi.getRun(runID1);
sigID1 = getSignalIDByIndex(run1,1);
sigID2 = getSignalIDByIndex(run1,2);

sig1 = Simulink.sdi.getSignal(sigID1);
sig2 = Simulink.sdi.getSignal(sigID2);

% Set tolerances for q, rad/sec
sig1.AbsTol = 0.1;
sig1.TimeTol = 0.6;

% Set tolerances for alpha, rad
sig2.AbsTol = 0.2;
sig2.TimeTol = 0.8;

% Run the comparison again
diffRun2 = Simulink.sdi.compareRuns(runID1,runID2);
sig1Result2 = getResultByIndex(diffRun2,1);
sig2Result2 = getResultByIndex(diffRun2,2);

% Check the result
sig1Result2.Status
ans = WithinTolerance

sig2Result2.Status
ans = WithinTolerance

 Alternatives

You can view and inspect comparison results using the Simulation Data Inspector UI. For more information, see “Compare Simulation Data”.

See Also
Simulink.sdi.DatasetRef.compare | Simulink.sdi.DiffSignalResult | Simulink.sdi.compareRuns | Simulink.sdi.compareSignals

Topics
“Inspect and Compare Data Programmatically”
“How the Simulation Data Inspector Compares Data”

Introduced in R2012b
get_result_by_index

**Class:** Simulink.sdi.DiffRunResult  
**Package:** Simulink.sdi

Return signal comparison result

**Syntax**

```matlab
diffSigObj = diffRunObj.getResultByIndex(index)
```

**Description**

diffSigObj = diffRunObj.getResultByIndex(index) returns the Simulink.sdi.DiffSignalResult object diffSigObj corresponding to the index in the Simulink.sdi.DiffRunResult object, diffRunObj.

**Input Arguments**

- **index** — Index of signal in run  
  integer

Index of the signal in the Simulink.sdi.DiffRunResult object.

**Output Arguments**

- **diffSigObj** — Simulink.sdi.DiffSignalResult object corresponding to the index  
  Simulink.sdi.DiffSignalResult object

Simulink.sdi.DiffSignalResult object for the signal at the specified index.
Examples

Analyze Simulation Data with Signal Tolerances

You can change tolerance values on a signal-by-signal basis to evaluate the effect of a model parameter change. This example uses the slexAircraftExample model and the Simulation Data Inspector to evaluate the effect of changing the time constant for the low-pass filter following the control input.

Setup

Load the model, and mark the \( q, \text{ rad/sec} \) and \( \alpha, \text{ rad} \) signals for logging. Then, simulate the model to create the baseline run.

```matlab
% Load example model
load_system('slexAircraftExample')

% Mark the \( q, \text{ rad/sec} \) and \( \alpha, \text{ rad} \) signals for logging
Simulink.sdi.markSignalForStreaming('slexAircraftExample/Aircraft Dynamics Model',3,'on')
Simulink.sdi.markSignalForStreaming('slexAircraftExample/Aircraft Dynamics Model',4,'on')

% Simulate system
out_1 = sim('slexAircraftExample');
```

Modify Model Parameter

Modify the model parameter \( Ts \) in the model workspace to change the time constant of the input low-pass filter.

```matlab
% Change input filter time constant
modelWorkspace = get_param('slexAircraftExample','modelworkspace');
assignin(modelWorkspace,'Ts',1)

% Simulate again
out_2 = sim('slexAircraftExample');
```

Compare Runs and Inspect Results

Use the Simulink.sdi.compareRuns function to compare the data from the simulations. Then, inspect the match property of the signal result to see whether the signals fell within the default tolerance of 0.

```matlab
% Get run data
runIDs = Simulink.sdi.getAllRunIDs;
```
runID1 = runIDs(end - 1);
runID2 = runIDs(end);

% Compare runs
diffRun1 = Simulink.sdi.compareRuns(runID1,runID2);

% Get signal result
sig1Result1 = getResultByIndex(diffRun1,1);
sig2Result1 = getResultByIndex(diffRun1,2);

% Check whether signals matched
sig1Result1.Status
ans =
OutOfRange

sig2Result1.Status
ans =
OutOfRange

**Compare Runs with Signal Tolerances**

The signals did not match within the default tolerance of 0. To further analyze the effect of the time constant change, add signal tolerances to the comparison with the baseline signal properties to determine the tolerance required for a pass. This example uses a combination of time and absolute tolerances.

% Get signal object for sigID1
run1 = Simulink.sdi.getRun(runID1);
sigID1 = getSignalIDByIndex(run1,1);
sigID2 = getSignalIDByIndex(run1,2);

sig1 = Simulink.sdi.getSignal(sigID1);
sig2 = Simulink.sdi.getSignal(sigID2);

% Set tolerances for q, rad/sec
sig1.AbsTol = 0.1;
sig1.TimeTol = 0.6;

% Set tolerances for alpha, rad
sig2.AbsTol = 0.2;
sig2.TimeTol = 0.8;

% Run the comparison again
diffRun2 = Simulink.sdi.compareRuns(runID1,runID2);
sig1Result2 = getResultByIndex(diffRun2,1);
sig2Result2 = getResultByIndex(diffRun2,2);

% Check the result
sig1Result2.Status
ans = WithinTolerance

sig2Result2.Status
ans = WithinTolerance

Alternatives
You can inspect comparison results using the Simulation Data Inspector UI. For more information, see “Compare Simulation Data”.

See Also
Simulink.sdi.DiffRunResult | Simulink.sdi.DiffSignalResult

Topics
“Inspect and Compare Data Programmatically”

Introduced in R2012b
Simulink.sdi.DiffSignalResult class

Package: Simulink.sdi

Access signal comparison results

Description

The Simulink.sdi.DiffSignalResult object provides access to the data and metadata created by a signal comparison. A Simulink.sdi.DiffSignalResult object gives access the difference signal, tolerance data, and the synchronized signal data.

Construction

DiffSignalObj = Simulink.sdi.compareSignals(signalID1, signalID2) creates a Simulink.sdi.DiffSignalResult object to provide access to the results of the comparison of the signals corresponding to sigID1 and sigID2.

DiffSignalObj = DiffRunObj.getResultByIndex(index) returns a Simulink.sdi.DiffSignalResult object for the signal comparison corresponding to the index within a Simulink.sdi.DiffRunResult object.

Input Arguments

signalID1 — Signal identifier for Baseline signal
integer

Numeric signal identifier for the Baseline signal in comparison.

signalID2 — Signal identifier for Compare to signal
integer

Numeric signal identifier for the Compare to signal in comparison.

index — Index of signal in run
integer
Index of the signal within the run.

Properties

Status — Comparison result signal status
WithinTolerance | OutOfTolerance | Unaligned | Pending | Processing | ...

Indicates the status of the DiffSignalResult object. The status can indicate where a given signal comparison is in a comparison process during a long comparison, or it can indicate information about the result of the signal comparison.

- **WithinTolerance** — Signal comparison completed, and all data points compared fell within the specified tolerance.
- **OutOfTolerance** — Signal comparison completed, and some data points compared fell outside of the specified tolerance.
- **Unaligned** — Signal from the **Baseline** run did not align with a signal in the **Compare to** run.
- **Empty** — Aligned **Baseline** or **Compare to** signal contains no data.
- **EmptySynced** — Synchronized **Baseline** or **Compare to** signal contains no data.
- **Canceled** — Signal result not computed because the user canceled the comparison or the algorithm ended the comparison before computing this signal result.
- **Pending** — Comparison is in progress and the signal result computation has not started.
- **Processing** — Signal result computation in progress.
- **UnitsMismatch** — **Baseline** and **Compare to** signal units do not match.
- **Unsupported** — Simulation Data Inspector comparison algorithm does not support this type of signal. For example, signals with data types that lose precision when converted to double are not supported.

For more information about alignment, tolerances, and synchronization, see “How the Simulation Data Inspector Compares Data”.

**SignalID1** — Baseline signal identifier

integer

Unique signal identifier for the **Baseline** comparison signal.
**SignalID2 — Compare to signal identifier**
integer

Unique signal identifier for the **Compare to** comparison signal.

**MaxDifference — Maximum difference**
double

Maximum difference between the two comparison signals.

**Sync1 — Synchronized Baseline signal**
timeseries

Synchronized **Baseline** signal. For more information about synchronization, see “How the Simulation Data Inspector Compares Data”.

**Sync2 — Synchronized Compare to signal**
timeseries

Synchronized **Compare to** signal. For more information about synchronization, see “How the Simulation Data Inspector Compares Data”.

**Diff — Difference signal**
timeseries

Difference signal resulting from the comparison as a timeseries object.

**Copy Semantics**

Handle. To learn how handle classes affect copy operations, see Copying Objects (MATLAB).

**Examples**

**Compare Signals Within a Simulation Run**

This example uses the `slexAircraftExample` model to demonstrate the comparison of the input and output signals for a control system. The example marks the signals for streaming then gets the run object for a simulation run. Signal IDs from the run object specify the signals to be compared.
% Load model slexAircraftExample and mark signals for streaming
load_system('slexAircraftExample')
Simulink.sdi.markSignalForStreaming('slexAircraftExample/Pilot',1,'on')
Simulink.sdi.markSignalForStreaming('slexAircraftExample/Aircraft Dynamics Model',4,'on')

% Simulate model slexAircraftExample
out = sim('slexAircraftExample');

% Get run IDs for most recent run
allIDs = Simulink.sdi.getAllRunIDs;
runID = allIDs(end);

% Get Run object
aircraftRun = Simulink.sdi.getRun(runID);

% Get signal IDs
signalID1 = getSignalIDByIndex(aircraftRun,1);
signalID2 = getSignalIDByIndex(aircraftRun,2);

if (isValidSignalID(aircraftRun,signalID1))
    % Change signal tolerance
    signal1 = Simulink.sdi.getSignal(signalID1);
    signal1.AbsTol = 0.1;
end

if (isValidSignalID(aircraftRun,signalID1) && isValidSignalID(aircraftRun,signalID2))
    % Compare signals
    sigDiff = Simulink.sdi.compareSignals(signalID1,signalID2);

    % Check whether signal result status
    match = sigDiff.Status
end

match = OutOfTolerance

Alternatives

You can view and inspect comparison results using the Simulation Data Inspector UI. For more information, see “Compare Simulation Data”.

Simulink.sdi.DiffSignalResult class

5-597
See Also
Simulink.sdi.DiffRunResult
Simulink.sdi.DiffRunResult.getResultByIndex | Simulink.sdi.compareRuns
| Simulink.sdi.compareSignals

Topics
“Inspect and Compare Data Programmatically”
“How the Simulation Data Inspector Compares Data”

Introduced in R2012b
Simulink.sdi.Run class

Package: Simulink.sdi

Access run signals and metadata

Description

The Simulink.sdi.Run object manages a run's metadata and the signals that comprise the run. You can use several methods to retrieve Simulink.sdi.Signal objects to access the signal data and metadata.

Construction

runObj = Simulink.sdi.Run.create creates an empty Simulink.sdi.Run object.

runObj = Simulink.sdi.getRun(runID) creates a Simulink.sdi.Run object, runObj, for the run corresponding to runID.

You can also use the Simulink.sdi.createRun and Simulink.sdi.createRunOrAddToStreamedRun functions to create Run objects.

Input Arguments

runID — Run identifier
integer

Unique number identifying the run.

Properties

id — Run identifier
integer

Unique numerical identification for the run.
**Name — Run name**

[] (default) | character vector

Name of the run. By default, `name` is empty.

Example: 'Run 1'

**Description — Run description**

[] (default) | character vector

Description of the run. By default, `Description` is empty.

Example: 'Initial simulation'

**Tag — Information tag**

[] (default) | character vector

Tag for additional run information. By default, `Tag` is empty. You can use the `Tag` parameter to categorize your simulation data or attach extra information to simulation runs.

Example: 'Gain = 2'

**DateCreated — Run creation time**

datetime object

Date and time the run was created.

Data Types: `datetime`

**SignalCount — Number of signals in run**

integer

Number of signals contained in the run.

**Model — Model that created the run**

character vector

Name of the model that created the run.

**SimMode — Simulation mode**

character vector

Simulation mode used to create the run, for runs created by simulation.
StartTime — Run start time
integer
First time point shared by all signals in the run.

StopTime — Run stop time
integer
Last time point shared by all signals in the run.

SLVersion — Simulink version used to create run
character vector
Version of Simulink used for the simulation that created the run.

ModelVersion — Model version used to create run
character vector
Version of the model simulated to create the run, taken from the Model Properties.

UserID — System account
character vector
System account used for the simulation that created the run. UserID only has a value for runs produced with Simulink simulations.

MachineName — Name of machine used for simulation
character vector
Name of the machine used for the simulation that created the run. MachineName only has a value for runs produced with Simulink simulations.

TaskName — Task name
[] (default) | character vector
Name of the simulation task that created the run for runs created with Parallel Computing Toolbox workers.

SolverType — Type of solver used to create run
'Variable-Step' | 'Fixed-Step'
The type of solver used for the simulation that created the run. SolverType only has a value for runs produced with Simulink simulations.
**SolverName — Name of solver used to create run**
character vector

Name of the solver used for the simulation that created the run. SolverName only has a value for runs produced with Simulink simulations.

Example: ode45

**ModelInitializationTime — Time to initialize model**
double

Amount of time to initialize the model for the simulation that created the run. ModelInitializationTime only has a value for runs produced with Simulink simulations.

**ModelExecutionTime — Time to execute model**
double

Execution time of the model simulation that created the run. ModelExecutionTime only has a value for runs produced with Simulink simulations.

**ModelTerminationTime — Time to terminate simulation**
double

Time to terminate the simulation that created the run. ModelTerminationTime only has a value for runs produced with Simulink simulations.

**ModelTotalElapsedTime — Total simulation time**
double

Total time to run model simulation that created the run. ModelTotalElapsedTime only has a value for runs produced with Simulink simulations.
Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>Add signals to run</td>
</tr>
<tr>
<td>create</td>
<td>Create a Simulink.sdi.Run object</td>
</tr>
<tr>
<td>export</td>
<td>Export run to base workspace or a file</td>
</tr>
<tr>
<td>getDatasetRef</td>
<td>Create a Simulink.sdi.DatasetRef object for a run</td>
</tr>
<tr>
<td>getSignalByIndex</td>
<td>Get Simulink.sdi.Signal object by index</td>
</tr>
<tr>
<td>getSignalIDByIndex</td>
<td>Return signal ID for signal at index</td>
</tr>
<tr>
<td>isValidSignalID</td>
<td>Determine whether signal ID is valid within a run</td>
</tr>
</tbody>
</table>

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects (MATLAB).

Examples

**Plot Signals from a Simulation Run**

This example demonstrates how to access the Simulink.sdi.Run object for a Simulation Data Inspector run created by logging signals. From the Simulink.sdi.Run object you can get Simulink.sdi.Signal objects that you can use to plot the data.

**Create a Simulation Run and Access the Run Object**

The sldemo_fuelsys model includes several signals that are marked for logging. To create a simulation run containing the logged data, simulate the model.

```
sim('ex_vdp');
```

The Simulation Data Inspector keeps track of runs by assigning each run created by simulation, importing data, or opening a session a unique numeric ID. To access the run object for the simulation just performed, use the Simulink.sdi.getAllRunIDs function and take the last run ID in the returned vector.

```
runIDs = Simulink.sdi.getAllRunIDs;
runID = runIDs(end);
```
Once you have the run ID for the run of interest, you can use the `Simulink.sdi.getRun` function to get the `Simulink.sdi.Run` object corresponding to the run. You can use the Run object to check metadata associated with the run, including the number of signals in the run.

```matlab
vdpRun = Simulink.sdi.getRun(runID);

vdpRun.SignalCount
ans = int32
    2
```

**Plot Data Using Signal Objects**

Use the `getSignalByIndex` function to access signals of interest from the Run object, `fuelRun`.

```matlab
signal1 = getSignalByIndex(vdpRun,1);
signal2 = getSignalByIndex(vdpRun,2);
```

Use the `Simulink.sdi.setSubPlotLayout` function to specify a 3-by-1 layout.

```matlab
Simulink.sdi.setSubPlotLayout(2,1)
```

Before plotting the data, use the `Simulink.sdi.clearAllSubPlots` function to clear any data that is already plotted.

```matlab
Simulink.sdi.clearAllSubPlots
```

Plot one signal on each subplot. To plot signals on the first subplot, you can set the `checked` property for the signal. To plot signals on subplots other than the first subplot, use the `plotOnSubPlot` function.

```matlab
signal1.Checked = true;
plotOnSubPlot(signal2,2,1,true);
```

**View the Plotted Data**

To view the plots you just created, open the Simulation Data Inspector using the `Simulink.sdi.view` function.
Create a Run and View the Data

This example shows how to create a run, add data to it, and then view the data in the Simulation Data Inspector.

Create Data for the Run

This example creates timeseries objects for a sine and a cosine. To visualize your data, the Simulation Data Inspector requires at least a time vector that corresponds with your data.

```matlab
% Generate timeseries data
time = linspace(0, 20, 100);
sine_vals = sin(2*pi/5*time);
```
sine_ts = timeseries(sine_vals, time);
sine_ts.Name = 'Sine, T=5';

cos_vals = cos(2*pi/8*time);
cos_ts = timeseries(cos_vals, time);
cos_ts.Name = 'Cosine, T=8';

Create a Simulation Data Inspector Run and Add Your Data

To give the Simulation Data Inspector access to your data, use the `create` method and create a run. This example modifies some of the run's properties to help identify the data. You can easily view run and signal properties with the Simulation Data Inspector.

```
% Create a run
sinusoidsRun = Simulink.sdi.Run.create;
sinusoidsRun.Name = 'Sinusoids';
sinusoidsRun.Description = 'Sine and cosine signals with different frequencies';

% Add timeseries data to run
sinusoidsRun.add('vars', sine_ts, cos_ts);
```

Plot Your Data Using the `Simulink.sdi.Signal` Object

The `getSignalByIndex` method returns a `Simulink.sdi.Signal` object that can be used to plot the signal in the Simulation Data Inspector. You can also programatically control aspects of the plot's appearance, such as the color and style of the line representing the signal. This example customizes the subplot layout and signal characteristics.

```
% Get signal, modify its properties, and change Checked property to true
sine_sig = sinusoidsRun.getSignalByIndex(1);
sine_sig.LineColor = [0 0 1];
sine_sig.LineDashed = '--';
sine_sig.Checked = true;

% Add another subplot for the cosine signal
Simulink.sdi.setSubPlotLayout(2, 1);

% Plot the cosine signal and customize its appearance
cos_sig = sinusoidsRun.getSignalByIndex(2);
cos_sig.LineColor = [0 1 0];
cos_sig.plotOnSubPlot(2, 1, true);

% View the signal in the Simulation Data Inspector
Simulink.sdi.view
```
Close the Simulation Data Inspector and Save Your Data

Simulink.sdi.close('sinusoids.mat')

Access Data from a Parallel Simulation

This example executes parallel simulations of the model slexAircraftExample with different input filter time constants and shows several ways to access the data using the Simulation Data Inspector programmatic interface.

Setup

Start by ensuring the Simulation Data Inspector is empty and Parallel Computing Toolbox support is configured to import runs created on local workers automatically. Then, create a vector of filter parameter values to use in each simulation.

% Make sure the Simulation Data Inspector is empty, and PCT support is % enabled.
Simulink.sdi.clear
Simulink.sdi.enablePCTSupport('local')

% Define Ts values
Ts_vals = [0.01, 0.02, 0.05, 0.1, 0.2, 0.5, 1];

Initialize Parallel Workers

Use gcp to create a pool of local workers to run parallel simulations if you don't already have one. In an spmd code block, load the slexAircraftExample model and select signals to log. To avoid data concurrency issues using sim in parfor, create a temporary directory for each worker to use during simulations.

p = gcp;

Starting parallel pool (parpool) using the 'local' profile ... connected to 4 workers.

spmd

% Load system and select signals to log
load_system('slexAircraftExample')
Simulink.sdi.markSignalForStreaming('slexAircraftExample/Pilot', 1, 'on')
Simulink.sdi.markSignalForStreaming('slexAircraftExample/Aircraft Dynamics Model',

% Create temporary directory on each worker
workDir = pwd;
addpath(workDir)
tempDir = tempname;
mkdir(tempDir)
cd(tempDir)
end

**Run Parallel Simulations**

Use `parfor` to run the seven simulations in parallel. Select the value for `Ts` for each simulation, and modify the value of `Ts` in the model workspace. Then, run the simulation and build an array of `Simulink.sdi.WorkerRun` objects to access the data with the Simulation Data Inspector. After the `parfor` loop, use another `spmd` segment to remove the temporary directories from the workers.

```
parfor index = 1:7
    % Select value for Ts
    Ts_val = Ts_vals(index);
    % Change the filter time constant and simulate
    modelWorkspace = get_param('slexAircraftExample','modelworkspace');
    modelWorkspace.assignin('Ts',Ts_val)
    sim('slexAircraftExample')
    % Create a worker run for each simulation
    workerRun(index) = Simulink.sdi.WorkerRun.getLatest
end

spmd
    % Remove temporary directories
    cd(workDir)
    rmdir(tempDir, 's')
    rmpath(workDir)
end
```

**Get Dataset Objects from Parallel Simulation Output**

The `getDataset` method puts the data from a `WorkerRun` into a `Dataset` object so you can easily post-process.
ds(7) = Simulink.SimulationData.Dataset;

for a = 1:7
    ds(a) = workerRun(a).getDataset;
end

ds(1)

ans =
Simulink.SimulationData.Dataset ' ' with 2 elements

<table>
<thead>
<tr>
<th>Name</th>
<th>BlockPath</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>[1x1 Signal] alpha, rad</td>
</tr>
<tr>
<td>2</td>
<td>[1x1 Signal] Stick</td>
</tr>
</tbody>
</table>

- Use braces { } to access, modify, or add elements using index.

Get DatasetRef Objects from Parallel Simulation Output

For big data workflows, use the getDatasetRef method to reference the data associated with the WorkerRun.

for b = 1:7
    datasetRef(b) = workerRun(b).getDatasetRef;
end

datasetRef(1)

ans =
DatasetRef with properties:

Name: 'Run 3: slexAircraftExample'
Run: [1x1 Simulink.sdi.Run]
numElements: 2

Process Parallel Simulation Data in the Simulation Data Inspector

You can also create local Run objects to analyze and visualize your data using the Simulation Data Inspector API. This example adds a tag indicating the filter time constant value for each run.

for c = 1:7
Runs(c) = workerRun(c).getLocalRun;
Ts_val_str = num2str(Ts_vals(c));
desc = strcat('Ts = ', Ts_val_str);
Runs(c).Description = desc;
Runs(c).Name = strcat('slexAircraftExample run Ts=', Ts_val_str);
end

Clean Up Worker Repositories

Clean up the files used by the workers to free up disk space for other simulations you want to run on your worker pool.

Simulink.sdi.cleanupWorkerResources

Alternatives

You can view runs and their properties in the Simulation Data Inspector UI. You can also import data to create runs in the Simulation Data Inspector GUI. For more information, see “View Data with the Simulation Data Inspector”.

See Also

Simulink.sdi.Signal | Simulink.sdi.WorkerRun |
Simulink.sdi.WorkerRun.getLocalRun | Simulink.sdi.addToRun |
Simulink.sdi.createRun | Simulink.sdi.getRun |
Simulink.sdi.getRunIDByIndex | Simulink.sdi.setRunNamingRule

Topics

“Inspect and Compare Data Programmatically”

Introduced in R2012b
add

Class: Simulink.sdi.Run  
Package: Simulink.sdi

Add signals to run

Syntax

run.add(sig)
run.add(source, filename)
run.add(source, ts1, ts2)

Description

run.add(sig) adds the data sig to the Simulink.sdi.Run object run from the base workspace.

run.add(source, filename) adds the data in the file filename to the Simulink.sdi.Run object, run.

run.add(source, ts1, ts2) allows you to add multiple signals to the run from the base workspace.

Input Arguments

sig — Data to add

Signals to add to the run. Data types that you can add to a run include:

- timeseries
- Simulink.SimulationData.Dataset
- Simulink.SimulationOutput
- timetable
• Data logged with Structure with time format
• Simscape variables

**source — Data source selector**

'value' | 'vars'

Source of the data to add to the run, specified as a character vector.

• 'value' indicates that the data comes from a file.
• 'vars' indicates that the data comes from one or more variables in the workspace.

**filename — File containing data**

character vector

File with data to add to the run.

Example: 'data.mat'

**ts1, ts2 — Workspace variables to add to run**

timeseries

Data to add to the run in one or more timeseries objects.

---

**Examples**

**Create a Run and View the Data**

This example shows how to create a run, add data to it, and then view the data in the Simulation Data Inspector.

**Create Data for the Run**

This example creates timeseries objects for a sine and a cosine. To visualize your data, the Simulation Data Inspector requires at least a time vector that corresponds with your data.

```matlab
% Generate timeseries data
time = linspace(0, 20, 100);
sine_vals = sin(2*pi/5*time);
sine_ts = timeseries(sine_vals, time);```
sine_ts.Name = 'Sine, T=5';

cos_vals = cos(2*pi/8*time);
cos_ts = timeseries(cos_vals, time);
cos_ts.Name = 'Cosine, T=8';

Create a Simulation Data Inspector Run and Add Your Data

To give the Simulation Data Inspector access to your data, use the `create` method and create a run. This example modifies some of the run's properties to help identify the data. You can easily view run and signal properties with the Simulation Data Inspector.

% Create a run
sinusoidsRun = Simulink.sdi.Run.create;
sinusoidsRun.Name = 'Sinusoids';
sinusoidsRun.Description = 'Sine and cosine signals with different frequencies';

% Add timeseries data to run
sinusoidsRun.add('vars', sine_ts, cos_ts);

Plot Your Data Using the Simulink.sdi.Signal Object

The `getSignalByIndex` method returns a Simulink.sdi.Signal object that can be used to plot the signal in the Simulation Data Inspector. You can also programmatically control aspects of the plot's appearance, such as the color and style of the line representing the signal. This example customizes the subplot layout and signal characteristics.

% Get signal, modify its properties, and change Checked property to true
sine_sig = sinusoidsRun.getSignalByIndex(1);
sine_sig.LineColor = [0 0 1];
sine_sig.LineDashed = '--';
sine_sig.Checked = true;

% Add another subplot for the cosine signal
Simulink.sdi.setSubPlotLayout(2, 1);

% Plot the cosine signal and customize its appearance
cos_sig = sinusoidsRun.getSignalByIndex(2);
cos_sig.LineColor = [0 1 0];
cos_sig.plotOnSubPlot(2, 1, true);

% View the signal in the Simulation Data Inspector
Simulink.sdi.view
Close the Simulation Data Inspector and Save Your Data

Simulink.sdi.close('sinusoids.mat')

See Also
Simulink.sdi.Run | Simulink.sdi.Run.create | Simulink.sdi.addToRun | Simulink.sdi.createRunOrAddToStreamedRun

Topics
“View Data with the Simulation Data Inspector”

Introduced in R2017b
Simulink.sdi.Run.create

**Class:** Simulink.sdi.Run  
**Package:** Simulink.sdi

Create a Simulink.sdi.Run object

**Syntax**

```matlab
runObj = Simulink.sdi.Run.create
```

**Description**

`runObj = Simulink.sdi.Run.create` creates the empty run object, `runObj`. You can add signals to the Run object with the `Simulink.sdi.Run.add` method or the `Simulink.sdi.addToRun` function. For more information on the `Simulink.sdi.Run` object and its properties, see `Simulink.sdi.Run`.

**Output Arguments**

- `runObj` — **Simulink.sdi.Run object**
  
  Simulink.sdi.Run object

  Empty Simulink.sdi.Run object.

**Examples**

**Create a Run and View the Data**

This example shows how to create a run, add data to it, and then view the data in the Simulation Data Inspector.
Create Data for the Run

This example creates timeseries objects for a sine and a cosine. To visualize your data, the Simulation Data Inspector requires at least a time vector that corresponds with your data.

% Generate timeseries data
time = linspace(0, 20, 100);
sine_vals = sin(2*pi/5*time);
sine_ts = timeseries(sine_vals, time);
sine_ts.Name = 'Sine, T=5';

cos_vals = cos(2*pi/8*time);
cos_ts = timeseries(cos_vals, time);
cos_ts.Name = 'Cosine, T=8';

Create a Simulation Data Inspector Run and Add Your Data

To give the Simulation Data Inspector access to your data, use the create method and create a run. This example modifies some of the run's properties to help identify the data. You can easily view run and signal properties with the Simulation Data Inspector.

% Create a run
sinusoidsRun = Simulink.sdi.Run.create;
sinusoidsRun.Name = 'Sinusoids';
sinusoidsRun.Description = 'Sine and cosine signals with different frequencies';

% Add timeseries data to run
sinusoidsRun.add('vars', sine_ts, cos_ts);

Plot Your Data Using the Simulink.sdi.Signal Object

The getSignalByIndex method returns a Simulink.sdi.Signal object that can be used to plot the signal in the Simulation Data Inspector. You can also programmatically control aspects of the plot's appearance, such as the color and style of the line representing the signal. This example customizes the subplot layout and signal characteristics.

% Get signal, modify its properties, and change Checked property to true
sine_sig = sinusoidsRun.getSignalByIndex(1);
sine_sig.LineColor = [0 0 1];
sine_sig.LineDashed = '-.';
sine_sig.Checked = true;
% Add another subplot for the cosine signal
Simulink.sdi.setSubPlotLayout(2, 1);

% Plot the cosine signal and customize its appearance
cos_sig = sinusoidsRun.getSignalByIndex(2);
cos_sig.LineColor = [0 1 0];
cos_sig.plotOnSubPlot(2, 1, true);

% View the signal in the Simulation Data Inspector
Simulink.sdi.view

Close the Simulation Data Inspector and Save Your Data
Simulink.sdi.close('sinusoids.mat')

See Also
Simulink.sdi.Run | Simulink.sdi.Run.add | Simulink.sdi.createRun | Simulink.sdi.createRunOrAddToStreamedRun

Topics
“Inspect and Compare Data Programmatically”

Introduced in R2017b
export

Class: Simulink.sdi.Run
Package: Simulink.sdi

Export run to base workspace or a file

Syntax

ds = runObj.export
runObj.export(Name,Value)

Description

ds = runObj.export exports the Simulink.sdi.Run object runObj to the base workspace in the Simulink.SimulationData.Dataset, ds.

runObj.export(Name,Value) exports the Simulink.sdi.Run object runObj to the base workspace or a file according to options specified by one or more name-value pair arguments.

Input Arguments

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name, Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside quotes. You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

Example: 'to','file'

to — Specify where to export data
'veariable' (default) | 'file'

Specify whether to export data to a variable in the base workspace or to a file.
When you export data to a file, you must also specify a file name using the 'filename' name-value pair argument. You can specify a file name with a MAT, MLDATX, or XLSX extension.

When you export a run to a MAT-file, the data is saved in a Simulink.SimulationData.Dataset object. When you export more than one run to a MAT-file using the Simulink.sdi.exportRun function, the data is saved in a Dataset object, where each element is a Dataset object containing the data for an exported run.

Data exported to a Microsoft Excel file is saved with the format described in “Simulation Data Inspector Import and Export File Format”.

When you export data to a Microsoft Excel file, you can specify additional options using the 'overwrite', 'metadata', and 'sharetimecolumn' name-value pairs.

Example: 'to', 'file'

**filename — Name of file to contain exported data**

string

Name of the file to contain the exported data, specified as a string or character array. Include a MAT, MLDATX, or XLSX extension in the file name to specify whether to export to a MAT-file, MLDATX-file, or a Microsoft Excel file. When you do not specify an extension with the file name, the data exports to a MAT-file.

Use the 'filename' name-value pair argument when you specify the 'to' name-value pair argument with the value 'file'.

When you export data to a Microsoft Excel file, you can specify additional options using the 'overwrite', 'metadata', and 'sharetimecolumn' name-value pair arguments.

Example: 'filename','mySpreadsheet.xlsx'

**overwrite — Specify data to overwrite in existing Microsoft Excel file**

'file' (default) | 'sheetsonly'

Specify data to overwrite in existing Microsoft Excel file. When you export data to a Microsoft Excel file that already exists, you can control whether the export overwrites the entire file or only the affected sheets in the file.

- 'file' — Overwrite the entire file with the exported data.
- 'sheetsonly' — Only overwrite sheets of the Microsoft Excel file with data that corresponds to the exported data.
Example: 'overwrite','sheetsonly'

**metadata** — Specify metadata to include in exported Microsoft Excel file

[] (default) | string array

Metadata to include in the exported Microsoft Excel file, specified in a string array. By default, the export operation does not include any metadata. You can export this metadata to the Microsoft Excel file:

- **dataType** — Signal data type
- **units** — Signal units
- **blockPath** — Path to the source block for logged signals
- **interp** — Signal interpolation method
- **portIndex** — Index of the port on the source block for logged signals

You can specify the desired metadata in any order you choose in the string array. The order of the metadata does not affect the format in the exported file, which always matches the description in “Simulation Data Inspector Import and Export File Format”.

Example: 'metadata',['units','dataType']

**sharetimecolumn** — Specify whether signals share time columns in exported Microsoft Excel file

'on' (default) | 'off'

Specify whether to share time columns in the exported Microsoft Excel file for signals that have identical time data. By default, signals with the same time data share a time column in the exported file. When you specify the value as 'off', each signal in the exported file has its own time column.

Example: 'sharetimecolumn','off'

**Output Arguments**

**ds** — Simulink.SimulationData.Dataset object containing run data

Simulink.SimulationData.Dataset object

Simulink.SimulationData.Dataset object containing the run data.
Examples

Export Run Data

This example shows how to export data from a run in the Simulation Data Inspector to a Simulink.SimulationData.Dataset object in the base workspace you can use to further process your data. The method you choose to export your run depends on the processing you do in your script. If you have a run object for the run, you can use the export method to create a Simulink.SimulationData.Dataset object with the run data in the base workspace. If you do not have a run object, use the Simulink.sdi.exportRun function to export the run to the workspace.

Export Run Using Simulink.sdi.exportRun

Use the Simulink.sdi.export function to export run data to the workspace or a file when your workflow does not include creating a run object.

To create a run of simulation data, open the vdp model, mark signals for logging, and run a simulation.

```matlab
load_system('vdp')

SignalHandles = get_param('vdp', 'Lines');

Simulink.sdi.markSignalForStreaming(SignalHandles(5).Handle, 'on')
Simulink.sdi.markSignalForStreaming(SignalHandles(6).Handle, 'on')

out = sim('vdp');
```

Use the Simulink.sdi.getAllRunIDs function to access the most recently created run.

```matlab
runIDs = Simulink.sdi.getAllRunIDs;
runID = runIDs(end);
```

Use the Simulink.sdi.exportRun function to export the run data to a Dataset object in the workspace.

```matlab
simDataset = Simulink.sdi.exportRun(runID);
```

Export Run Using export Method

When your workflow involves creating a Run object, you can use the export function to create a Simulink.SimulationData.Dataset object in the base workspace for further
processing of the run data. For example, you need to access Run objects for simulation runs with signal data you want to compare using the `Simulink.sdi.compareSignals` function.

Load a model and mark signals for logging. Then, simulate the model to create run data.

```matlab
load_system('vdp')
SignalHandles = get_param('vdp', 'Lines');
Simulink.sdi.markSignalForStreaming(SignalHandles(5).Handle, 'on')
Simulink.sdi.markSignalForStreaming(SignalHandles(6).Handle, 'on')
sim('vdp');
```

Use the `Simulink.sdi.getAllRunIDs` function to access the run ID for the most recently created run. Then, use the `Simulink.sdi.getRun` function to access the Run object corresponding to the run.

```matlab
runIDs = Simulink.sdi.getAllRunIDs;
runID = runIDs(end);
vdpNet = Simulink.sdi.getRun(runID);
```

Use the `export` function to export the run data to a `Dataset` object in the workspace.

```matlab
simDataset = export(vdpRun);
```

**Alternatives**

You can export run data programmatically using the `Simulink.sdi.exportRun` function, or you can use the Simulation Data Inspector UI. For more information, see “Save and Share Simulation Data Inspector Data and Views”.

**See Also**

`Simulink.sdi.Run` | `Simulink.sdi.exportRun` | `Simulink.sdi.getRun` | `Simulink.sdi.report`

**Topics**

“Inspect and Compare Data Programmatically”
“Save and Share Simulation Data Inspector Data and Views”

Introduced in R2017b
getDatasetRef

Class: Simulink.sdi.Run
Package: Simulink.sdi

Create a Simulink.sdi.DatasetRef object for a run

Syntax

DatasetRef = runObj.getDatasetRef

Description

DatasetRef = runObj.getDatasetRef creates a Simulink.sdi.DatasetRef object with the data in the Simulink.sdi.Run object, runObj.

Output Arguments

DatasetRef — Simulink.sdi.DatasetRef object
Simulink.sdi.DatasetRef object

Simulink.sdi.DatasetRef object that provides access to the run data.

Examples

Compare Runs with the Simulink.sdi.DatasetRef Object

This example shows how to work with the Simulink.sdi.DatasetRef object by comparing two runs of the ex_sldemo_absbrake system with different desired slip ratios.

% Simulate model ex_sldemo_absbrake to create a run of logged signals
load_system('ex_sldemo_absbrake')
sim('ex_sldemo_absbrake')
% Get the runID
runIDs = Simulink.sdi.getAllRunIDs;
runID = runIDs(end);

% Get the run object
brakeRun = Simulink.sdi.getRun(runID);

% Make a Simulink.sdi.DatasetRef object
run_DSRef = brakeRun.getDatasetRef;

% Get the names of the elements in the object
names = run_DSRef.getElementNames

names = 2x1 cell array
   {'yout'}
   {'slp' }

% Get yout bus
[yout, name, index] = run_DSRef.getElement(1);

% View signals in outputs
outputs = yout.Values

outputs = struct with fields:
   Ww: [1x1 timeseries]
   Vs: [1x1 timeseries]
   Sd: [1x1 timeseries]

% Get slp signal
slp = run_DSRef.getSignal('slp');

% Plot signal
slp.Checked = 'true';

% Create another run for a different Desired relative slip
set_param('ex_sldemo_absbrake/Desired relative slip', 'Value', '0.25')
sim('ex_sldemo_absbrake')
DSR_Runs = Simulink.sdi.DatasetRef;

% Compare the results from the two runs
[matches, mismatches, diffResult] = run_DSRef.compare(DSR_Runs(2));
% Open the Simulation Data Inspector to view signals
run_DSRef.plot

Alternatives

You can also create a `Simulink.sdi.DatasetRef` object using the `Simulink.sdi.DatasetRef` constructor.

See Also

`Simulink.sdi.DatasetRef` | `Simulink.sdi.Run`

Topics

“Inspect and Compare Data Programmatically”

Introduced in R2017b
getSignalByIndex

Class: Simulink.sdi.Run
Package: Simulink.sdi

Get Simulink.sdi.Signal object by index

Syntax

signal = runObj.getSignalByIndex(index)

Description

signal = runObj.getSignalByIndex(index) returns a signal object for the signal at the specified index within the Simulink.sdi.Run object, runObj.

Input Arguments

index — Signal index
integer

Index of the signal within the run.

Output Arguments

signal — Simulink.sdi.Signal object
Simulink.sdi.Signal object

Simulink.sdi.Signal object for the signal at the specified index in the run.
Examples

Plot Signals from a Simulation Run

This example demonstrates how to access the Simulink.sdi.Run object for a Simulation Data Inspector run created by logging signals. From the Simulink.sdi.Run object you can get Simulink.sdi.Signal objects that you can use to plot the data.

Create a Simulation Run and Access the Run Object

The sldemo_fuelsys model includes several signals that are marked for logging. To create a simulation run containing the logged data, simulate the model.

```matlab
sim('ex_vdp');
```

The Simulation Data Inspector keeps track of runs by assigning each run created by simulation, importing data, or opening a session a unique numeric ID. To access the run object for the simulation just performed, use the Simulink.sdi.getAllRunIDs function and take the last run ID in the returned vector.

```matlab
runIDs = Simulink.sdi.getAllRunIDs;
runID = runIDs(end);
```

Once you have the run ID for the run of interest, you can use the Simulink.sdi.getRun function to get the Simulink.sdi.Run object corresponding to the run. You can use the Run object to check metadata associated with the run, including the number of signals in the run.

```matlab
vdpRun = Simulink.sdi.getRun(runID);
vdpRun.SignalCount
```

```
ans = int32
  2
```

Plot Data Using Signal Objects

Use the getSignalByIndex function to access signals of interest from the Run object, fuelRun.

```matlab
signal1 = getSignalByIndex(vdpRun,1);
signal2 = getSignalByIndex(vdpRun,2);
```

Use the Simulink.sdi.setSubPlotLayout function to specify a 3-by-1 layout.
Simulink.sdi.setSubPlotLayout(2,1)

Before plotting the data, use the `Simulink.sdi.clearAllSubPlots` function to clear any data that is already plotted.

`Simulink.sdi.clearAllSubPlots`

Plot one signal on each subplot. To plot signals on the first subplot, you can set the checked property for the signal. To plot signals on subplots other than the first subplot, use the `plotOnSubPlot` function.

```matlab
signal1.Checked = true;
plotOnSubPlot(signal2,2,1,true);
```

**View the Plotted Data**

To view the plots you just created, open the Simulation Data Inspector using the `Simulink.sdi.view` function.
Alternatives

You can access signal properties, view signals, and export data to the workspace using the Simulation Data Inspector UI. For more information, see “Organize Your Simulation Data Inspector Workspace”.

See Also
Topics
“Inspect and Compare Data Programmatically”

Introduced in R2012b
getSignalIDByIndex

Class: Simulink.sdi.Run
Package: Simulink.sdi

Return signal ID for signal at index

Syntax

signalID = runObj.getSignalIDByIndex(index)

Description

signalID = runObj.getSignalIDByIndex(index) returns the signal ID for the signal at the specified index in the Simulink.sdi.Run object. You can use the signal ID to create a Simulink.sdi.Signal object or to perform a signal comparison with Simulink.sdi.compareSignals.

Input Arguments

index — Signal index
integer

Index of the signal within the run.

Output Arguments

signalID — Signal identifier
integer

Unique numeric signal identifier.
Examples

Compare Signals Within a Simulation Run

This example uses the `slexAircraftExample` model to demonstrate the comparison of the input and output signals for a control system. The example marks the signals for streaming then gets the run object for a simulation run. Signal IDs from the run object specify the signals to be compared.

```matlab
% Load model slexAircraftExample and mark signals for streaming
load_system('slexAircraftExample')
Simulink.sdi.markSignalForStreaming('slexAircraftExample/Pilot',1,'on')
Simulink.sdi.markSignalForStreaming('slexAircraftExample/Aircraft Dynamics Model',4,'on')

% Simulate model slexAircraftExample
out = sim('slexAircraftExample');

% Get run IDs for most recent run
allIDs = Simulink.sdi.getAllRunIDs;
runID = allIDs(end);

% Get Run object
aircraftRun = Simulink.sdi.getRun(runID);

% Get signal IDs
signalID1 = getSignalIDByIndex(aircraftRun,1);
signalID2 = getSignalIDByIndex(aircraftRun,2);

if (isValidSignalID(aircraftRun,signalID1))
    % Change signal tolerance
    signal1 = Simulink.sdi.getSignal(signalID1);
    signal1.AbsTol = 0.1;
end

if (isValidSignalID(aircraftRun,signalID1) && isValidSignalID(aircraftRun,signalID2))
    % Compare signals
    sigDiff = Simulink.sdi.compareSignals(signalID1,signalID2);

    % Check whether signal result status
    match = sigDiff.Status
end

match = OutOfTolerance
```
Alternatives

You can access signal properties, view signals, and export data to the workspace using the Simulation Data Inspector UI. For more information, see “Organize Your Simulation Data Inspector Workspace”.

See Also

Topics
“Inspect and Compare Data Programmatically”

Introduced in R2012b
isValidSignalID

Class: Simulink.sdi.Run
Package: Simulink.sdi

Determine whether signal ID is valid within a run

Syntax

isValid = runObj.isValidSignalID(signalID)

Description

isValid = runObj.isValidSignalID(signalID) returns a logical indication of whether the signalID corresponds to a signal in the Simulink.sdi.Run object runObj.

Input Arguments

signalID — Signal identifier
integer

Unique numeric signal identifier.

Output Arguments

isValid — Logical indicator
logical

Logical indicator of signal ID validity.

- true when the signalID corresponds to a signal in the run object.
- false when the signalID does not correspond to a signal in the run object.
Examples

Compare Signals Within a Simulation Run

This example uses the slexAircraftExample model to demonstrate the comparison of the input and output signals for a control system. The example marks the signals for streaming then gets the run object for a simulation run. Signal IDs from the run object specify the signals to be compared.

```matlab
% Load model slexAircraftExample and mark signals for streaming
load_system('slexAircraftExample')
Simulink.sdi.markSignalForStreaming('slexAircraftExample/Pilot',1,'on')
Simulink.sdi.markSignalForStreaming('slexAircraftExample/Aircraft Dynamics Model',4,'on')

% Simulate model slexAircraftExample
out = sim('slexAircraftExample');

% Get run IDs for most recent run
allIDs = Simulink.sdi.getAllRunIDs;
runID = allIDs(end);

% Get Run object
aircraftRun = Simulink.sdi.getRun(runID);

% Get signal IDs
signalID1 = getSignalIDByIndex(aircraftRun,1);
signalID2 = getSignalIDByIndex(aircraftRun,2);

if (isValidSignalID(aircraftRun,signalID1))
    % Change signal tolerance
    signal1 = Simulink.sdi.getSignal(signalID1);
    signal1.AbsTol = 0.1;
end

if (isValidSignalID(aircraftRun,signalID1) && isValidSignalID(aircraftRun,signalID2))
    % Compare signals
    sigDiff = Simulink.sdi.compareSignals(signalID1,signalID2);

    % Check whether signal result status
    match = sigDiff.Status
end

match = OutOfTolerance
```
See Also
Simulink.sdi.Run | Simulink.sdi.Run.getSignalIDByIndex | Simulink.sdi.Signal | Simulink.sdi.compareSignals

Topics
“Inspect and Compare Data Programmatically”

Introduced in R2012b
Simulink.sdi.Signal class

Package: Simulink.sdi

Access signal data and metadata

Description

A Simulink.sdi.Signal object provides access to signal data and metadata. The metadata includes properties for visualizing and comparing signals.

Construction

signal = Simulink.sdi.getSignal(signalID) returns a Simulink.sdi.Signal object corresponding to the signalID.

signal = runObj.getSignal(signalID) returns a Simulink.sdi.Signal object corresponding to the signalID.

signal = runObj.getSignalByIndex(index) returns a Simulink.sdi.Signal object corresponding to the signalID at the index specified by index in the Run object runObj.

Input Arguments

signalID — Signal identifier
integer

Numeric signal identifier for the signal generated by the Simulation Data Inspector. You can get the signal ID for a signal using methods of the Simulink.sdi.Run object or using the Simulink.sdi.getSignal function.

index — Signal index within run
integer

Index of the signal within the run.
Example: 1

Properties

Signal Properties

ID — Signal identifier
integer

Unique number identifying the signal. The Simulation Data Inspector assigns a unique numeric ID to each run and signal.

RunID — Run identifier
integer

Run identifier for the run that contains the signal. The Simulation Data Inspector assigns a unique numeric ID to each run and signal. You can access the run ID for a signal using the Simulink.sdi.getAllRunIDs function.

Name — Signal name
character vector

Name of the signal.
Example: 'fuel'

Description — Signal description
'' (default) | character array | string

Description of the signal, specified as a character array or a string. You can use the Description property to annotate the signal or identify the signal content beyond the Name. When you log Simscape data to the Simulation Data Inspector, the Description property is populated automatically for each node.

Units — Signal measurement units
character vector

Signal units of measure.
Example: 'g/s'
**Data Type — Data type for signal data**
class vector

Data type of signal data.
Example: 'double'

**Complexity — Complexity of signal data**
class vector

Specifies whether signal data is real or complex.
Example: 'real'

**SampleTime — Signal sample time**
class vector

Signal sample time. A value of 'Continuous' indicates a variable-step simulation.
Example: 'Continuous'
Example: 0.1

**Model — Name of model that produced signal**
class vector

Name of the model that produced the signal.
Example: 'sldemo_fuelsys'

**BlockPath — Path of block that defines signal**
class vector

The path to the block that produced the signal.
Example: 'sldemo_fuelsys/Engine Gas Dynamics'

**FullBlockPath — Path of the block that defines the signal**
class vector

Path to the block that generates the signal including the full model hierarchy. For signals within reference models, FullBlockPath is a cell array containing the full path. For other signals, FullBlockPath is identical to BlockPath.
Example: 'sldemo_fuelsys/Engine Gas Dynamics/Mixing & Combustion/MinMax'
**PortIndex** — Block port index
integer

Index of the output port that defines the signal.
Example: 1

**Dimensions** — Dimensions of matrix containing signal
integer array

Dimensions of the matrix that contains the signal.
Example: [1]

**Channel** — Index of signal within matrix
integer array

Indices of the signal for signals that are part of a vector or matrix.

**Values** — Signal values
timeseries

Time and data values for the signal. For buses, Values is a structure.

**RootSource** — High-level logging structure containing signal imported from workspace
character vector

Name of the high-level logging structure containing the signal, for signals imported from the MATLAB workspace.

**TimeSource** — Source of signal time data imported from workspace
character vector

Name of the variable containing the signal time data for signals imported from the MATLAB workspace.

**DataSource** — Source of data imported from workspace
character vector

Name of the array containing the signal data for signals imported from the MATLAB workspace.
Visualization Properties

ComplexFormat — Display format for complex signals
"real-imaginary" | "magnitude" | "magnitude-phase" | "phase"

Complex format specifying how to display complex signal data in the Simulation Data Inspector.

- "real-imaginary" — The real and imaginary components of the signal display together when you plot the signal. The imaginary component of the signal is plotted with a different shade of the Line Color.
- "magnitude" — The magnitude of the signal displays when you plot the signal.
- "magnitude-phase" — The magnitude and phase of the signal display together when you plot the signal.
- "phase" — The phase of the signal displays when you plot the signal. The phase is plotted with a different shade of the Line Color.

Data Types: char | string

Checked — Plotting indicator
false (default) | true

Logical value indicating whether the signal is plotted on any subplot. Setting Checked to false clears the signal from all subplots. Setting Checked to true plots the signal on the active subplot.

Data Types: logical

LineColor — Signal line color
1-by-3 vector

Color of signal in plots, specified as a 1-by-3 RGB vector.

Example: [0 114 189]

Data Types: double

LineDashed — Signal line style
'-' | '--' | ':' | '-.'

Signal line style.
• ' - ' specifies a solid line style.
• ' - - ' specifies a dashed line style.
• ':' specifies a dotted line style.
•'-. ' specifies a dash-dot line style.

**InterpMethod — Interpolation method**

'linear' (default) | 'zoh' | 'none'

Interpolation method used in data visualization and synchronization. 'zoh' specifies zero-order hold interpolation, 'linear' specifies linear interpolation, and 'none' specifies no interpolation. For more information about the interpolation options, see “How the Simulation Data Inspector Compares Data”.

**Comparison Properties**

**AbsTol — Absolute tolerance**

0 (default) | double

Positive-valued absolute tolerance of the signal used for signal comparisons. The Simulation Data Inspector uses tolerances specified in the signal properties of the baseline signal when **Override Global Tol** is set to **yes**. For more information about tolerances in the Simulation Data Inspector, see “How the Simulation Data Inspector Compares Data”.

Example: 0.1

Data Types: double

**RelTol — Relative tolerance**

0 (default) | double

Positive-valued relative tolerance for the signal used for signal comparisons. The Simulation Data Inspector uses tolerances specified in the signal properties of the baseline signal when **Override Global Tol** is set to **yes**. The relative tolerance is expressed as a fractional multiplier. For example, 0.1 specifies a 10 percent tolerance. For more information about tolerances in the Simulation Data Inspector, see “How the Simulation Data Inspector Compares Data”.

Example: 0.05

Data Types: double
TimeTol — Time tolerance

0 (default) | double

Positive-valued time tolerance for the signal used in signal comparisons. The Simulation Data Inspector uses tolerances specified in the signal properties of the baseline signal when Override Global Tol is set to yes. Specify the time tolerance in seconds. For more information about tolerances in the Simulation Data Inspector, see “How the Simulation Data Inspector Compares Data”.

Example: 0.1

Data Types: double

SyncMethod — Synchronization method

'union' (default) | 'intersection'

Method used to synchronize signal time data for comparison. For more information about the synchronization options, see “How the Simulation Data Inspector Compares Data”.

Methods

convertUnits Convert signal units
event Export signal object to MATLAB timeseries
getAsTall Return tall timetable with time and data values
plotOnSubPlot Plot signal on specified sub-plot

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects (MATLAB).

Examples

Analyze Simulation Data with Signal Tolerances

You can change tolerance values on a signal-by-signal basis to evaluate the effect of a model parameter change. This example uses the slexAircraftExample model and the
Simulation Data Inspector to evaluate the effect of changing the time constant for the low-pass filter following the control input.

**Setup**

Load the model, and mark the $q$, rad/sec and $\alpha$, rad signals for logging. Then, simulate the model to create the baseline run.

```matlab
% Load example model
load_system('slexAircraftExample')

% Mark the $q$, rad/sec and $\alpha$, rad signals for logging
Simulink.sdi.markSignalForStreaming('slexAircraftExample/Aircraft Dynamics Model',3,'on')
Simulink.sdi.markSignalForStreaming('slexAircraftExample/Aircraft Dynamics Model',4,'on')

% Simulate system
out_1 = sim('slexAircraftExample');
```

**Modify Model Parameter**

Modify the model parameter $Ts$ in the model workspace to change the time constant of the input low-pass filter.

```matlab
% Change input filter time constant
modelWorkspace = get_param('slexAircraftExample','modelworkspace');
assignin(modelWorkspace,'Ts',1)

% Simulate again
out_2 = sim('slexAircraftExample');
```

**Compare Runs and Inspect Results**

Use the Simulink.sdi.compareRuns function to compare the data from the simulations. Then, inspect the match property of the signal result to see whether the signals fell within the default tolerance of 0.

```matlab
% Get run data
runIDs = Simulink.sdi.getAllRunIDs;
runID1 = runIDs(end-1);
runID2 = runIDs(end);

% Compare runs
diffRun1 = Simulink.sdi.compareRuns(runID1,runID2);

% Get signal result
```
sig1Result1 = getResultByIndex(diffRun1,1);
sig2Result1 = getResultByIndex(diffRun1,2);

% Check whether signals matched
sig1Result1.Status
ans =
OutOfTolerance

sig2Result1.Status
ans =
OutOfTolerance

**Compare Runs with Signal Tolerances**

The signals did not match within the default tolerance of 0. To further analyze the effect of the time constant change, add signal tolerances to the comparison with the baseline signal properties to determine the tolerance required for a pass. This example uses a combination of time and absolute tolerances.

% Get signal object for sigID1
run1 = Simulink.sdi.getRun(runID1);
sigID1 = getSignalIDByIndex(run1,1);
sigID2 = getSignalIDByIndex(run1,2);

sig1 = Simulink.sdi.getSignal(sigID1);
sig2 = Simulink.sdi.getSignal(sigID2);

% Set tolerances for q, rad/sec
sig1.AbsTol = 0.1;
sig1.TimeTol = 0.6;

% Set tolerances for alpha, rad
sig2.AbsTol = 0.2;
sig2.TimeTol = 0.8;

% Run the comparison again
diffRun2 = Simulink.sdi.compareRuns(runID1,runID2);
sig1Result2 = getResultByIndex(diffRun2,1);
sig2Result2 = getResultByIndex(diffRun2,2);

% Check the result
sig1Result2.Status
ans =  
WithinTolerance  
sig2Result2.Status  
ans =  
WithinTolerance

**Define Comparison and Visualization Properties for a Signal**

This example shows how to obtain a `Simulink.sdi.Signal` object and modify its properties using the Simulation Data Inspector programmatic interface.

**Acquire a Simulink.sdi.Signal Object**

First, run a simulation to create a run. This example uses example model `slexAircraftExample`. Then, use the Simulation Data Inspector programmatic interface to get the `Simulink.sdi.Signal` object for your signal of interest.

```matlab
% Configure model "slexAircraftExample" for logging and simulate
simOut = sim('slexAircraftExample','SaveOutput','on',...
    'SaveFormat','StructureWithTime');

% Use Simulink.sdi.createRun to create a run and return the list of signal IDs for signals contained in the run
[~,~,signalIDs] = Simulink.sdi.createRun('My Run','base',{simOut});

% Get the signal object corresponding to the first signal ID
signalObj = Simulink.sdi.getSignal(signalIDs(1));
```

**Modify the Signal Properties**

The `Simulink.sdi.Signal` object has several comparison and visualization properties that you can modify.

```matlab
% Define comparison and visualization properties for this signal
signalObj.syncMethod = 'intersection';
signalObj.lineColor = [1,0.4,0.6];
signalObj.lineDashed = '-';
signalObj.checked = true;
```

**View the Signal Properties**

You can view the signal properties in the command window and in the Simulation Data Inspector to verify that the signal has its properties defined how you want them.
signalObj

signalObj =
    Signal with properties:
        ID: 268273
    RunID: 268259
        Name: 'Integrate:CSTATE'
    Description: ""
    Units: ''
    DataType: 'double'
    Complexity: "real"
    ComplexFormat: "real-imaginary"
    SampleTime: ''
    Model: 'slexAircraftExample'
    BlockPath: 'slexAircraftExample/Aircraft Dynamics Model/Vertical Channel/Integrate'
    FullBlockPath: 'slexAircraftExample/Aircraft Dynamics Model/Vertical Channel/Integrate'
    PortIndex: 0
    Dimensions: 1
        Channel: [1x0 int32]
        Checked: 1
    LineColor: [1 0.4000 0.6000]
    LineDashed: '-'
    InterpMethod: 'linear'
        AbsTol: 0
        RelTol: 0
        TimeTol: 0
    SyncMethod: 'intersection'
        Values: [1x1 timeseries]
    RootSource: 'simOut.get('xout')'
    TimeSource: 'simOut.get('xout').time'
    DataSource: 'simOut.get('xout').signals(1).values'

Simulink.sdi.view

**Compare Signals Within a Simulation Run**

This example uses the `slexAircraftExample` model to demonstrate the comparison of the input and output signals for a control system. The example marks the signals for streaming then gets the run object for a simulation run. Signal IDs from the run object specify the signals to be compared.

% Load model `slexAircraftExample` and mark signals for streaming
load_system('slexAircraftExample')
Simulink.sdi.markSignalForStreaming('slexAircraftExample/Pilot',1,'on')
Simulink.sdi.markSignalForStreaming('slexAircraftExample/Aircraft Dynamics Model',4,'on')

% Simulate model slexAircraftExample
out = sim('slexAircraftExample');

% Get run IDs for most recent run
allIDs = Simulink.sdi.getAllRunIDs;
runID = allIDs(end);

% Get Run object
aircraftRun = Simulink.sdi.getRun(runID);

% Get signal IDs
signalID1 = getSignalIDByIndex(aircraftRun,1);
signalID2 = getSignalIDByIndex(aircraftRun,2);

if (isValidSignalID(aircraftRun,signalID1))
    % Change signal tolerance
    signal1 = Simulink.sdi.getSignal(signalID1);
    signal1.AbsTol = 0.1;
end

if (isValidSignalID(aircraftRun,signalID1) && isValidSignalID(aircraftRun,signalID2))
    % Compare signals
    sigDiff = Simulink.sdi.compareSignals(signalID1,signalID2);

    % Check whether signal result status
    match = sigDiff.Status
end

match =
OutOfTolerance

**Plot Signals from a Simulation Run**

This example demonstrates how to access the Simulink.sdi.Run object for a Simulation Data Inspector run created by logging signals. From the Simulink.sdi.Run object you can get Simulink.sdi.Signal objects that you can use to plot the data.

**Create a Simulation Run and Access the Run Object**

The sldemo_fuelsys model includes several signals that are marked for logging. To create a simulation run containing the logged data, simulate the model.
The Simulation Data Inspector keeps track of runs by assigning each run created by simulation, importing data, or opening a session a unique numeric ID. To access the run object for the simulation just performed, use the `Simulink.sdi.getAllRunIDs` function and take the last run ID in the returned vector.

```matlab
runIDs = Simulink.sdi.getAllRunIDs;
runID = runIDs(end);
```

Once you have the run ID for the run of interest, you can use the `Simulink.sdi.getRun` function to get the `Simulink.sdi.Run` object corresponding to the run. You can use the `Run` object to check metadata associated with the run, including the number of signals in the run.

```matlab
vdpRun = Simulink.sdi.getRun(runID);
```

```matlab
vdpRun.SignalCount
ans = int32
 2
```

### Plot Data Using Signal Objects

Use the `getSignalByIndex` function to access signals of interest from the Run object, `fuelRun`.

```matlab
signal1 = getSignalByIndex(vdpRun,1);
signal2 = getSignalByIndex(vdpRun,2);
```

Use the `Simulink.sdi.setSubPlotLayout` function to specify a 3-by-1 layout.

```matlab
Simulink.sdi.setSubPlotLayout(2,1)
```

Before plotting the data, use the `Simulink.sdi.clearAllSubPlots` function to clear any data that is already plotted.

```matlab
Simulink.sdi.clearAllSubPlots
```

Plot one signal on each subplot. To plot signals on the first subplot, you can set the `checked` property for the signal. To plot signals on subplots other than the first subplot, use the `plotOnSubPlot` function.

```matlab
signal1.Checked = true;
plotOnSubPlot(signal2,2,1,true);
```
**View the Plotted Data**

To view the plots you just created, open the Simulation Data Inspector using the `Simulink.sdi.view` function.

**Alternatives**

Use the Simulation Data Inspector UI to view and modify signals and signal properties.
See Also
Simulink.sdi.DatasetRef.getSignal | Simulink.sdi.Run |
Simulink.sdi.Run.getSignalByIndex |
Simulink.sdi.Run.getSignalIDByIndex | Simulink.sdi.createRun |
Simulink.sdi.getSignal

Topics
“Inspect and Compare Data Programmatically”
“How the Simulation Data Inspector Compares Data”

Introduced in R2012b
**convertUnits**

**Class:** Simulink.sdi.Signal  
**Package:** Simulink.sdi

Convert signal units

**Syntax**

```matlab
signal.convertUnits(units)
```

**Description**

`signal.convertUnits(units)` converts the units of `signal` to the units specified by `units`. For a list of acceptable units, see Allowed Units. You can use `Simulink.sdi.Signal.convertUnits` to convert the units on `Simulink.sdi.Signal` objects with data of all built-in and fixed-point types.

**Input Arguments**

- **units** — Desired signal units  
  string | character vector

  Desired units for the signal.

  Example: 'm'  
  Example: "ft/s"

  Data Types: char | string
Examples

Programmatically Convert Signal Units

This example shows how to use the `convertUnits` method to convert the units of a `Simulink.sdi.Signal` object, using the model `sldemo_autotrans`.

Generate Simulation Data

Simulate the model to create a run of data. Then, use the Simulation Data Inspector programmatic interface to get the run data.

```matlab
% Simulate the model
sim('sldemo_autotrans')

% Get a Simulink.sdi.Run object for the most recently created run
ids = Simulink.sdi.getAllRunIDs;
id = ids(end);
transRun = Simulink.sdi.getRun(id);
```

Inspect the Signal Properties

Get a `Simulink.sdi.Signal` object for the `EngineRPM` signal and inspect its properties to determine the units.

```matlab
% Get Simulink.sdi.Signal object
signal = transRun.getSignalByIndex(1)

signal =
Signal with properties:
   ID: 63757
   RunID: 63751
   Name: 'EngineRPM'
   Description: ""
   Units: 'rpm'
   DataType: 'double'
   Complexity: "real"
   ComplexFormat: "real-imaginary"
   SampleTime: '0.04'
   Model: 'sldemo_autotrans'
   BlockPath: 'sldemo_autotrans/Engine'
   FullBlockPath: 'sldemo_autotrans/Engine'
   PortIndex: 1
```
Convert Signal Units

Use the convertUnits method to convert the EngineRPM signal units to rad/s. Then, change the signal name to reflect the new units.

```matlab
% Convert units
signal.convertUnits('rad/s')

signal.Name = 'Engine,rad/s';
```

See Also
Simulink.sdi.Signal

Topics
“Inspect and Compare Data Programmatically”
“Unit Specification in Simulink Models”
“Units in Simulink”

Introduced in R2018a
export

Class: Simulink.sdi.Signal
Package: Simulink.sdi

Export signal object to MATLAB timeseries

Syntax

ts = sigObj.export
ts = sigObj.export(startTime, endTime)
sigObj.export( ___ ,Name,Value)

Description

ts = sigObj.export exports the Simulink.sdi.Signal object sigObj to the timeseries ts in the MATLAB workspace.

ts = sigObj.export(startTime, endTime) exports the portion of the Simulink.sdi.Signal object defined by startTime and endTime to the timeseries ts in the MATLAB workspace.

sigObj.export( ___ ,Name,Value) exports the Simulink.sdi.Signal object to the base workspace or a file according to the options specified by one or more name-value pair arguments.

Input Arguments

startTime — Export start time
integer

Start time for the signal portion to export.

Example: 0

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64
**endTime — Export end time**

integer

End time for the signal portion to export.

Example: 10

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

**Name-Value Pair Arguments**

Specify optional comma-separated pairs of Name, Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside quotes. You can specify several name and value pair arguments in any order as Name1, Value1, ..., NameN, ValueN.

Example: 'to', 'file'

**to — Specify where to export data**

'variable' (default) | 'file'

Specify whether to export data to a variable in the base workspace or to a file.

When you export data to a file, you must also specify a file name using the 'filename' name-value pair argument. You can specify a file name with a MAT or XLSX extension.

When you export a signal to a MAT-file, the data is saved in the MAT-file as a timeseries object.

Data exported to a Microsoft Excel file is saved with the format described in “Simulation Data Inspector Import and Export File Format”.

When you export data to a Microsoft Excel file, you can specify additional options using the 'overwrite', 'metadata', and 'sharetimecolumn' name-value pairs.

Example: 'to', 'file'

**filename — Name of file to contain exported data**

string

Name of the file to contain the exported data, specified as a string or character array. Include a MAT or XLSX extension in the file name to specify whether to export to a MAT-
file or a Microsoft Excel file. When you do not specify an extension with the file name, the data exports to a MAT-file.

Use the 'filename' name-value pair argument when you specify the 'to' name-value pair argument with the value 'file'.

When you export data to a Microsoft Excel file, you can specify additional options using the 'overwrite', 'metadata', and 'sharetimecolumn' name-value pair arguments.

Example: 'filename', "mySpreadsheet.xlsx"

overwrite — Specify data to overwrite in existing file

'file' (default) | 'sheetsonly'

Specify data to overwrite in existing file. When you export data to a Microsoft Excel file that already exists, you can control whether the export overwrites the entire file or only the affected sheets in the file.

- 'file' — Overwrite the entire file with the exported data.
- 'sheetsonly' — Only overwrite sheets of the Microsoft Excel file with data that corresponds to the exported data.

Example: 'overwrite', 'sheetsonly'

metadata — Specify metadata to include in exported Microsoft Excel file

[] (default) | string array

Metadata to include in the exported Microsoft Excel file, specified in a string array. By default, the export operation does not include any metadata. You can export this metadata to the Microsoft Excel file:

- dataType — Signal data type
- units — Signal units
- blockPath — Path to the source block for logged signals
- interp — Signal interpolation method
- portIndex — Index of the port on the source block for logged signals

You can specify the desired metadata in any order you choose in the string array. The order of the metadata does not affect the format in the exported file, which always matches the description in “Simulation Data Inspector Import and Export File Format”.

Example: 'metadata', ["units", "dataType"]
sharetimecolumn — Specify whether signals share time columns in exported file
'on' (default) | 'off'

Specify whether to share time columns in the exported file for signals that have identical
time data. By default, signals with the same time data share a time column in the
exported file. When you specify the value as 'off', each signal in the exported file has its
own time column.

Example: 'sharetimecolumn','off'

Output Arguments

ts — Exported timeseries
timeseries

Timeseries exported from Simulink.sdi.Signal object.

Examples

Export Signal Data to Timeseries

This example shows how to generate a signal and export the signal data to a timeseries.

% Simulate model sldemo_fuelsys to create a run of logged signals
sim('sldemo_fuelsys');

% Get the runID
runIDs = Simulink.sdi.getAllRunIDs;
runID = runIDs(end);

% Get the run object
fuelRun = Simulink.sdi.getRun(runID);

% Get a signal
signal = fuelRun.getSignalByIndex(2);

% Export signal data to timeseries
ts = signal.export;
Alternatives

You can export data to MATLAB or a MAT-file from the Simulation Data Inspector UI. For more information, see “Save and Share Simulation Data Inspector Data and Views”.

See Also
Simulink.sdi.Signal | Simulink.sdi.exportRun

Topics
“Inspect and Compare Data Programmatically”
“Save and Share Simulation Data Inspector Data and Views”

Introduced in R2017b
getAsTall

Class: Simulink.sdi.Signal
Package: Simulink.sdi

Return tall timetable with time and data values

Syntax

tt = sigObj.getAsTall

Description

tt = sigObj.getAsTall returns a tall timetable of the time and data values in the Simulink.sdi.Signal object sigObj. For more information on working with tall arrays, see “Tall Arrays for Out-of-Memory Data” (MATLAB).

Output Arguments

**tt — Tall timetable**
tall timetable

Tall timetable containing the data from the Simulink.sdi.Signal object.

Examples

Get Tall Timetable of Signal Data

This example shows how to generate a tall timetable from signal data in a Simulink.sdi.Signal object.

```matlab
% Simulate the model sldemo_fuelsys to create a run of logged signals
sim('sldemo_fuelsys');
```
% Get the runID
runIDs = Simulink.sdi.getAllRunIDs;
runID = runIDs(end);

% Get the run object
fuelRun = Simulink.sdi.getRun(runID);

% Get a signal
signal = fuelRun.getSignalByIndex(2);

% Get tall timetable of signal
tt = signal.getAsTall

tt =

Mx1 tall timetable

<table>
<thead>
<tr>
<th>Time</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 sec</td>
<td>0.068493</td>
</tr>
<tr>
<td>0.00056199 sec</td>
<td>0.092452</td>
</tr>
<tr>
<td>0.0033719 sec</td>
<td>0.21101</td>
</tr>
<tr>
<td>0.01 sec</td>
<td>0.48273</td>
</tr>
<tr>
<td>0.02 sec</td>
<td>0.88522</td>
</tr>
<tr>
<td>0.03 sec</td>
<td>1.2763</td>
</tr>
<tr>
<td>0.04 sec</td>
<td>1.6563</td>
</tr>
<tr>
<td>0.05 sec</td>
<td>2.0255</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

See Also
Simulink.sdi.Signal | Simulink.sdi.Signal.export | Simulink.sdi.exportRun | tall

Topics
“Inspect and Compare Data Programmatically”

Introduced in R2017b
plotOnSubPlot

Class: Simulink.sdi.Signal
Package: Simulink.sdi

Plot signal on specified sub-plot

Syntax

signalObj.plotOnSubplot(r,c,checked)

Description

signalObj.plotOnSubplot(r,c,checked) plots or clears the signal corresponding to the Simulink.sdi.Signal object, sigObj, on the sub-plot specified by r and c.

Input Arguments

r — Row index
scalar

Row index for the sub-plot.
Example: 1

c — Column index
scalar

Column index for the sub-plot.
Example: 2

checked — Signal checked parameter state
true | false

Desired state for signal on sub-plot.
• true plots the signal on the sub-plot.
• false clears the signal from the sub-plot.

Data Types: logical

Examples

Plot Signals from a Simulation Run

This example demonstrates how to access the Simulink.sdi.Run object for a Simulation Data Inspector run created by logging signals. From the Simulink.sdi.Run object you can get Simulink.sdi.Signal objects that you can use to plot the data.

Create a Simulation Run and Access the Run Object

The sldemo_fuelsys model includes several signals that are marked for logging. To create a simulation run containing the logged data, simulate the model.

sim('ex_vdp');

The Simulation Data Inspector keeps track of runs by assigning each run created by simulation, importing data, or opening a session a unique numeric ID. To access the run object for the simulation just performed, use the Simulink.sdi.getAllRunIDs function and take the last run ID in the returned vector.

runIDs = Simulink.sdi.getAllRunIDs;
runID = runIDs(end);

Once you have the run ID for the run of interest, you can use the Simulink.sdi.getRun function to get the Simulink.sdi.Run object corresponding to the run. You can use the Run object to check metadata associated with the run, including the number of signals in the run.

vdpRun = Simulink.sdi.getRun(runID);

vdpRun.SignalCount

ans = int32
     2
Plot Data Using Signal Objects

Use the `getSignalByIndex` function to access signals of interest from the Run object, `fuelRun`.

```matlab
signal1 = getSignalByIndex(vdpRun,1);
signal2 = getSignalByIndex(vdpRun,2);
```

Use the `Simulink.sdi.setSubPlotLayout` function to specify a 3-by-1 layout.

```matlab
Simulink.sdi.setSubPlotLayout(2,1)
```

Before plotting the data, use the `Simulink.sdi.clearAllSubPlots` function to clear any data that is already plotted.

```matlab
Simulink.sdi.clearAllSubPlots
```

Plot one signal on each subplot. To plot signals on the first subplot, you can set the checked property for the signal. To plot signals on subplots other than the first subplot, use the `plotOnSubPlot` function.

```matlab
signal1.Checked = true;
plotOnSubPlot(signal2,2,1,true);
```

View the Plotted Data

To view the plots you just created, open the Simulation Data Inspector using the `Simulink.sdi.view` function.
Alternatives

You can use the Simulation Data Inspector GUI to modify your plot layout and where you plot signals. For more information, see “Inspect Simulation Data”.

See Also

Simulink.sdi.Signal | Simulink.sdi.clearAllSubPlots | Simulink.sdi.setSubPlotLayout
Topics
“Inspect and Compare Data Programmatically”
“Create Plots Using the Simulation Data Inspector”

Introduced in R2017b
Simulink.sdi.WorkerRun class

**Package:** Simulink.sdi

Access simulation data from parallel workers

**Description**

The Simulink.sdi.WorkerRun class provides access to run data generated on Parallel Computing Toolbox parallel workers. Create a Simulink.sdi.WorkerRun object on the worker, and then use the object to access data in your local MATLAB session.

**Construction**

```matlab
workerRun = Simulink.sdi.WorkerRun(runID)
```

creates a Simulink.sdi.WorkerRun object with the run identifier specified by `runID`.

```matlab
workerRun = Simulink.sdi.WorkerRun.getLatest
```

creates a Simulink.sdi.WorkerRun object of the most recent run.

**Input Arguments**

- `runID` — Run identifier
  
  integer

  Run identifier

**Methods**

- `getDataset`
  
  Create Dataset of worker run data

- `getDatasetRef`
  
  Create DatasetRef for worker run

- `getLatest`
  
  Create worker run for latest run

- `getLocalRun`
  
  Create local run from worker run
Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects (MATLAB).

Examples

Access Data from a Parallel Simulation

This example executes parallel simulations of the model slexAircraftExample with different input filter time constants and shows several ways to access the data using the Simulation Data Inspector programmatic interface.

Setup

Start by ensuring the Simulation Data Inspector is empty and Parallel Computing Toolbox support is configured to import runs created on local workers automatically. Then, create a vector of filter parameter values to use in each simulation.

```matlab
% Make sure the Simulation Data Inspector is empty, and PCT support is enabled.
Simulink.sdi.clear
Simulink.sdi.enablePCTSupport('local')

% Define Ts values
Ts_vals = [0.01, 0.02, 0.05, 0.1, 0.2, 0.5, 1];
```

Initialize Parallel Workers

Use gcp to create a pool of local workers to run parallel simulations if you don’t already have one. In an `spmd` code block, load the slexAircraftExample model and select signals to log. To avoid data concurrency issues using `sim` in `parfor`, create a temporary directory for each worker to use during simulations.

```matlab
p = gcp;

Starting parallel pool (parpool) using the 'local' profile ... connected to 4 workers.

spmd

    % Load system and select signals to log
```
load_system('slexAircraftExample')
Simulink.sdi.markSignalForStreaming('slexAircraftExample/Pilot', 1, 'on')
Simulink.sdi.markSignalForStreaming('slexAircraftExample/Aircraft Dynamics Model', 4, 'on')

% Create temporary directory on each worker
workDir = pwd;
addpath(workDir)
tempDir = tempname;
mkdir(tempDir)
cd(tempDir)

end

Run Parallel Simulations

Use parfor to run the seven simulations in parallel. Select the value for Ts for each simulation, and modify the value of Ts in the model workspace. Then, run the simulation and build an array of Simulink.sdi.WorkerRun objects to access the data with the Simulation Data Inspector. After the parfor loop, use another spmd segment to remove the temporary directories from the workers.

parfor index = 1:7

    % Select value for Ts
    Ts_val = Ts_vals(index);

    % Change the filter time constant and simulate
    modelWorkspace = get_param('slexAircraftExample','modelworkspace');
    modelWorkspace.assignin('Ts',Ts_val)
sim('slexAircraftExample')

    % Create a worker run for each simulation
    workerRun(index) = Simulink.sdi.WorkerRun.getLatest

end

spmd

    % Remove temporary directories
    cd(workDir)
    rmdir(tempDir, 's')
rmpath(workDir)
end
Get Dataset Objects from Parallel Simulation Output

The `getDataset` method puts the data from a `WorkerRun` into a `Dataset` object so you can easily post-process.

\[
\text{ds}(7) = \text{Simulink.SimulationData.Dataset};
\]

\[
\text{for } a = 1:7 \\
\quad \text{ds}(a) = \text{workerRun}(a).\text{getDataset};
\]

\[
\text{ds}(1)
\]

ans =
\[
\text{Simulink.SimulationData.Dataset} ' ' \text{with 2 elements}
\]

<table>
<thead>
<tr>
<th>Name</th>
<th>BlockPath</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 [1x1 Signal]</td>
<td>alpha, rad</td>
</tr>
<tr>
<td>2 [1x1 Signal]</td>
<td>Stick</td>
</tr>
</tbody>
</table>

- Use braces { } to access, modify, or add elements using index.

Get DatasetRef Objects from Parallel Simulation Output

For big data workflows, use the `getDatasetRef` method to reference the data associated with the `WorkerRun`.

\[
\text{for } b = 1:7 \\
\quad \text{datasetRef}(b) = \text{workerRun}(b).\text{getDatasetRef};
\]

\[
\text{datasetRef}(1)
\]

ans =
\[
\text{DatasetRef with properties:}
\]

- Name: 'Run 3: slexAircraftExample'
- Run: [1x1 Simulink.sdi.Run]
- numElements: 2
Process Parallel Simulation Data in the Simulation Data Inspector

You can also create local Run objects to analyze and visualize your data using the Simulation Data Inspector API. This example adds a tag indicating the filter time constant value for each run.

```plaintext
for c = 1:7
    Runs(c) = workerRun(c).getLocalRun;
    Ts_val_str = num2str(Ts_vals(c));
    desc = strcat('Ts = ', Ts_val_str);
    Runs(c).Description = desc;
    Runs(c).Name = strcat('slexAircraftExample run Ts=', Ts_val_str);
end
```

Clean Up Worker Repositories

Clean up the files used by the workers to free up disk space for other simulations you want to run on your worker pool.

Simulink.sdi.cleanupWorkerResources

Alternatives

You can also access, view, and analyze simulation data from Parallel Computing Toolbox workers using the Simulation Data Inspector UI.

See Also

Simulink.sdi.Run

Topics

“Inspect and Compare Data Programmatically”

Introduced in R2017b
**getDataset**

**Class:** Simulink.sdi.WorkerRun  
**Package:** Simulink.sdi

Create Dataset of worker run data

**Syntax**

```plaintext
dataset = workerRun.getDataset
dataset = workerRun.getDataset(domain)
```

**Description**

```plaintext
dataset = workerRun.getDataset returns a
Simulink.SimulationData.Dataset object of the data corresponding to the
Simulink.sdi.WorkerRun object.

dataset = workerRun.getDataset(domain) returns a
Simulink.SimulationData.Dataset object of the data corresponding to the
Simulink.sdi.WorkerRun object limited to the scope specified by domain.
```

**Input Arguments**

```plaintext
domain — Scope specifier
'signals' | 'outports'
```

Scope limiting argument that selects the data to return in the
Simulink.SimulationData.Dataset object.

- 'signals' limits the data returned in the Dataset to signals in the WorkerRun.
- 'outports' limits the data returned in the Dataset to outport data in the WorkerRun.
Output Arguments

dataset — Simulink.SimulationData.Dataset object
Simulink.SimulationData.Dataset object

Simulink.SimulationData.Dataset object containing the data from the Simulink.sdi.WorkerRun object.

Examples

Access Data from a Parallel Simulation

This example executes parallel simulations of the model slexAircraftExample with different input filter time constants and shows several ways to access the data using the Simulation Data Inspector programmatic interface.

Setup

Start by ensuring the Simulation Data Inspector is empty and Parallel Computing Toolbox support is configured to import runs created on local workers automatically. Then, create a vector of filter parameter values to use in each simulation.

```matlab
% Make sure the Simulation Data Inspector is empty, and PCT support is enabled.
Simulink.sdi.clear
Simulink.sdi.enablePCTSupport('local')
```

% Define Ts values
Ts_vals = [0.01, 0.02, 0.05, 0.1, 0.2, 0.5, 1];

Initialize Parallel Workers

Use gcp to create a pool of local workers to run parallel simulations if you don't already have one. In an spmd code block, load the slexAircraftExample model and select signals to log. To avoid data concurrency issues using sim in parfor, create a temporary directory for each worker to use during simulations.

```matlab
p = gcp;
```

Starting parallel pool (parpool) using the 'local' profile ...
connected to 4 workers.
spmd

% Load system and select signals to log
load_system('slexAircraftExample')
Simulink.sdi.markSignalForStreaming('slexAircraftExample/Pilot', 1, 'on')
Simulink.sdi.markSignalForStreaming('slexAircraftExample/Aircraft Dynamics Model', 4, 'on')

% Create temporary directory on each worker
workDir = pwd;
addpath(workDir)
tempDir = tempname;
mkdir(tempDir)

cd(tempDir)
end

Run Parallel Simulations

Use parfor to run the seven simulations in parallel. Select the value for Ts for each simulation, and modify the value of Ts in the model workspace. Then, run the simulation and build an array of Simulink.sdi.WorkerRun objects to access the data with the Simulation Data Inspector. After the parfor loop, use another spmd segment to remove the temporary directories from the workers.

parfor index = 1:7

% Select value for Ts
Ts_val = Ts_vals(index);

% Change the filter time constant and simulate
modelWorkspace = get_param('slexAircraftExample','modelworkspace');
modelWorkspace.assignin('Ts',Ts_val)
sim('slexAircraftExample')

% Create a worker run for each simulation
workerRun(index) = Simulink.sdi.WorkerRun.getLatest
end

spmd

% Remove temporary directories
cd(workDir)
rmdir(tempDir, 's')
Get Dataset Objects from Parallel Simulation Output

The `getDataset` method puts the data from a `WorkerRun` into a `Dataset` object so you can easily post-process.

```matlab
ds(7) = Simulink.SimulationData.Dataset;
for a = 1:7
    ds(a) = workerRun(a).getDataset;
end
ans =
Simulink.SimulationData.Dataset '' with 2 elements

    Name        BlockPath
    ________  ______________________________________
      1  [1x1 Signal]      alpha, rad  ...rcraftExample/Aircraft Dynamics Model
      2  [1x1 Signal]      Stick       slexAircraftExample/Pilot

- Use braces `{ }` to access, modify, or add elements using index.

Get DatasetRef Objects from Parallel Simulation Output

For big data workflows, use the `getDatasetRef` method to reference the data associated with the `WorkerRun`.

```matlab
for b = 1:7
datasetRef(b) = workerRun(b).getDatasetRef;
end
datasetRef(1)
ans =
    DatasetRef with properties:
    Name: 'Run 3: slexAircraftExample'
    Run: [1x1 Simulink.sdi.Run]
    numElements: 2
Process Parallel Simulation Data in the Simulation Data Inspector

You can also create local Run objects to analyze and visualize your data using the Simulation Data Inspector API. This example adds a tag indicating the filter time constant value for each run.

```matlab
for c = 1:7
    Runs(c) = workerRun(c).getLocalRun;
    Ts_val_str = num2str(Ts_vals(c));
    desc = strcat('Ts = ', Ts_val_str);
    Runs(c).Description = desc;
    Runs(c).Name = strcat('slexAircraftExample run Ts=', Ts_val_str);
end

Clean Up Worker Repositories

Clean up the files used by the workers to free up disk space for other simulations you want to run on your worker pool.

Simulink.sdi.cleanupWorkerResources

See Also

Topics
“Inspect and Compare Data Programmatically”

Introduced in R2017b
getDatasetRef

**Class:** Simulink.sdi.WorkerRun

**Package:** Simulink.sdi

Create DatasetRef for worker run

**Syntax**

```plaintext
datasetRef = workerRun.getDatasetRef
datasetRef = workerRun.getDatasetRef(domain)
```

**Description**

`datasetRef = workerRun.getDatasetRef` returns a `Simulink.SimulationData.DatasetRef` object referencing the data in the `Simulink.sdi.WorkerRun` object, `workerRun`.

`datasetRef = workerRun.getDatasetRef(domain)` returns a `Simulink.SimulationData.DatasetRef` object referencing the data in the `Simulink.sdi.WorkerRun` object, `workerRun`, limited to the scope specified by `domain`.

**Input Arguments**

**domain** — Scope limiting input

- `'signals'` | `'outports'`

Scope limiting argument that selects the data to reference in the `Simulink.SimulationData.DatasetRef` object.

- `'signals'` limits the data referenced in the `DatasetRef` to signals in the `WorkerRun`.
- `'outports'` limits the data referenced in the `DatasetRef` to outport data in the `WorkerRun`.  

5-678
Output Arguments

datasetRef — Simulink.SimulationData.DatasetRef object
Simulink.sdi.DatasetRef object

Simulink.sdi.DatasetRef object referencing the data in the Parallel Computing Toolbox worker run.

Examples

Access Data from a Parallel Simulation

This example executes parallel simulations of the model slexAircraftExample with different input filter time constants and shows several ways to access the data using the Simulation Data Inspector programmatic interface.

Setup

Start by ensuring the Simulation Data Inspector is empty and Parallel Computing Toolbox support is configured to import runs created on local workers automatically. Then, create a vector of filter parameter values to use in each simulation.

```matlab
% Make sure the Simulation Data Inspector is empty, and PCT support is % enabled.
Simulink.sdi.clear
Simulink.sdi.enablePCTSupport('local')

% Define Ts values
Ts_vals = [0.01, 0.02, 0.05, 0.1, 0.2, 0.5, 1];
```

Initialize Parallel Workers

Use gcp to create a pool of local workers to run parallel simulations if you don't already have one. In an spmd code block, load the slexAircraftExample model and select signals to log. To avoid data concurrency issues using sim in parfor, create a temporary directory for each worker to use during simulations.

```matlab
p = gcp;

Starting parallel pool (parpool) using the 'local' profile ... connected to 4 workers.
```
spmd

% Load system and select signals to log
load_system('slexAircraftExample')
Simulink.sdi.markSignalForStreaming('slexAircraftExample/Pilot', 1, 'on')
Simulink.sdi.markSignalForStreaming('slexAircraftExample/Aircraft Dynamics Model', 4, 'on')

% Create temporary directory on each worker
workDir = pwd;
addpath(workDir)
tempDir = tempname;
mkdir(tempDir)
cd(tempDir)
end

Run Parallel Simulations

Use parfor to run the seven simulations in parallel. Select the value for Ts for each simulation, and modify the value of Ts in the model workspace. Then, run the simulation and build an array of Simulink.sdi.WorkerRun objects to access the data with the Simulation Data Inspector. After the parfor loop, use another spmd segment to remove the temporary directories from the workers.

parfor index = 1:7

% Select value for Ts
Ts_val = Ts_vals(index);

% Change the filter time constant and simulate
modelWorkspace = get_param('slexAircraftExample','modelworkspace');
modelWorkspace.assignin('Ts',Ts_val)
sim('slexAircraftExample')

% Create a worker run for each simulation
workerRun(index) = Simulink.sdi.WorkerRun.getLatest
end

spmd

% Remove temporary directories
cd(workDir)
rmdir(tempDir, 's')

Get Dataset Objects from Parallel Simulation Output

The `getDataset` method puts the data from a `WorkerRun` into a `Dataset` object so you can easily post-process.

```matlab
ds(7) = Simulink.SimulationData.Dataset;
for a = 1:7
ds(a) = workerRun(a).getDataset;
end
ds(1)
ans =
Simulink.SimulationData.Dataset '' with 2 elements

<table>
<thead>
<tr>
<th>Name</th>
<th>BlockPath</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 [1x1 Signal] alpha, rad</td>
<td>...rcraftExample/Aircraft Dynamics Model</td>
</tr>
<tr>
<td>2 [1x1 Signal] Stick</td>
<td>slexAircraftExample/Pilot</td>
</tr>
</tbody>
</table>

- Use braces {} to access, modify, or add elements using index.

Get DatasetRef Objects from Parallel Simulation Output

For big data workflows, use the `getDatasetRef` method to reference the data associated with the `WorkerRun`.

```matlab
for b = 1:7
datasetRef(b) = workerRun(b).getDatasetRef;
end
datasetRef(1)
ans =
DatasetRef with properties:

    Name: 'Run 3: slexAircraftExample'
    Run: [1x1 Simulink.sdi.Run]
    numElements: 2
Process Parallel Simulation Data in the Simulation Data Inspector

You can also create local Run objects to analyze and visualize your data using the Simulation Data Inspector API. This example adds a tag indicating the filter time constant value for each run.

```matlab
for c = 1:7
    Runs(c) = workerRun(c).getLocalRun;
    Ts_val_str = num2str(Ts_vals(c));
    desc = strcat('Ts = ', Ts_val_str);
    Runs(c).Description = desc;
    Runs(c).Name = strcat('slexAircraftExample run Ts=', Ts_val_str);
end
```

Clean Up Worker Repositories

Clean up the files used by the workers to free up disk space for other simulations you want to run on your worker pool.

```matlab
Simulink.sdi.cleanupWorkerResources
```

See Also

- `Simulink.SimulationData.DatasetRef`
- `Simulink.sdi.WorkerRun.getDataset`
- `Simulink.sdi.WorkerRun.getLatest`
- `Simulink.sdi.WorkerRun.getLocalRun`

Topics

- “Inspect and Compare Data Programmatically”

Introduced in R2017b
Simulink.sdi.WorkerRun.getLatest

Class: Simulink.sdi.WorkerRun
Package: Simulink.sdi

Create worker run for latest run

Syntax

runObj = Simulink.sdi.WorkerRun.getLatest

Description

runObj = Simulink.sdi.WorkerRun.getLatest creates a Simulink.sdi.WorkerRun object for the latest run on a Parallel Computing Toolbox worker.

Output Arguments

runObj — Local Simulink.sdi.Run object
Simulink.sdi.WorkerRun object

Simulink.sdi.WorkerRun object to access the data from the latest Parallel Computing Toolbox worker run.

Examples

Access Data from a Parallel Simulation

This example executes parallel simulations of the model slexAircraftExample with different input filter time constants and shows several ways to access the data using the Simulation Data Inspector programmatic interface.
Setup

Start by ensuring the Simulation Data Inspector is empty and Parallel Computing Toolbox support is configured to import runs created on local workers automatically. Then, create a vector of filter parameter values to use in each simulation.

% Make sure the Simulation Data Inspector is empty, and PCT support is % enabled.
Simulink.sdi.clear
Simulink.sdi.enablePCTSupport('local')

% Define Ts values
Ts_vals = [0.01, 0.02, 0.05, 0.1, 0.2, 0.5, 1];

Initialize Parallel Workers

Use gcp to create a pool of local workers to run parallel simulations if you don't already have one. In an spmd code block, load the slexAircraftExample model and select signals to log. To avoid data concurrency issues using sim in parfor, create a temporary directory for each worker to use during simulations.

p = gcp;

Starting parallel pool (parpool) using the 'local' profile ... connected to 4 workers.

spmd

% Load system and select signals to log
load_system('slexAircraftExample')
Simulink.sdi.markSignalForStreaming('slexAircraftExample/Pilot', 1, 'on')
Simulink.sdi.markSignalForStreaming('slexAircraftExample/Aircraft Dynamics Model', 4, 'on')

% Create temporary directory on each worker
workDir = pwd;
addpath(workDir)
tempDir = tempname;
mkdir(tempDir)
cd(tempDir)
end
Run Parallel Simulations

Use `parfor` to run the seven simulations in parallel. Select the value for `Ts` for each simulation, and modify the value of `Ts` in the model workspace. Then, run the simulation and build an array of `Simulink.sdi.WorkerRun` objects to access the data with the Simulation Data Inspector. After the `parfor` loop, use another `spmd` segment to remove the temporary directories from the workers.

\[
\text{parfor } \text{index} = 1:7 \\
\quad \text{Ts\_val} = \text{Ts\_vals(index)}; \\
\quad \text{modelWorkspace} = \text{get\_param('slexAircraftExample','modelworkspace')};
\quad \text{modelWorkspace}\.\text{assignin('Ts','Ts\_val')} \\
\quad \text{sim('slexAircraftExample')} \\
\quad \text{workerRun(index)} = \text{Simulink.sdi.WorkerRun.getLatest}
\]

\[
\text{spmd} \\
\quad \text{cd(workDir)} \\
\quad \text{rmdir(tempDir, 's')} \\
\quad \text{rmpath(workDir)}
\]

Get Dataset Objects from Parallel Simulation Output

The `getDataset` method puts the data from a `WorkerRun` into a `Dataset` object so you can easily post-process.

\[
\text{ds(7)} = \text{Simulink.SimulationData.Dataset}; \\
\text{for } a = 1:7 \\
\quad \text{ds(a)} = \text{workerRun(a)}\.\text{getDataset}; \\
\text{end}
\]

\[
\text{ds(1)}
\]
ans =
Simulink.SimulationData.Dataset '' with 2 elements

<table>
<thead>
<tr>
<th>Name</th>
<th>BlockPath</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 [1x1 Signal]</td>
<td>alpha, rad  ...rcraftExample/Aircraft Dynamics Model</td>
</tr>
<tr>
<td>2 [1x1 Signal]</td>
<td>Stick slexAircraftExample/Pilot</td>
</tr>
</tbody>
</table>

- Use braces {} to access, modify, or add elements using index.

### Get DatasetRef Objects from Parallel Simulation Output

For big data workflows, use the getDatasetRef method to reference the data associated with the WorkerRun.

```matlab
for b = 1:7
    datasetRef(b) = workerRun(b).getDatasetRef;
end
```

datasetRef(1)

ans =
DatasetRef with properties:
    Name: 'Run 3: slexAircraftExample'
    Run: [1x1 Simulink.sdi.Run]
    numElements: 2

### Process Parallel Simulation Data in the Simulation Data Inspector

You can also create local Run objects to analyze and visualize your data using the Simulation Data Inspector API. This example adds a tag indicating the filter time constant value for each run.

```matlab
for c = 1:7

    Runs(c) = workerRun(c).getLocalRun;
    Ts_val_str = num2str(Ts_vals(c));
    desc = strcat('Ts = ', Ts_val_str);
    Runs(c).Description = desc;
    Runs(c).Name = strcat('slexAircraftExample run Ts=', Ts_val_str);

end
```
Clean Up Worker Repositories

Clean up the files used by the workers to free up disk space for other simulations you want to run on your worker pool.

Simulink.sdi.cleanupWorkerResources

See Also

Simulink.sdi.WorkerRun | Simulink.sdi.WorkerRun.getDataset | Simulink.sdi.WorkerRun.getDatasetRef | Simulink.sdi.WorkerRun.getLocalRun

Topics

“Inspect and Compare Data Programmatically”

Introduced in R2017b
getLocalRun

Class: Simulink.sdi.WorkerRun
Package: Simulink.sdi

Create local run from worker run

Syntax

runObj = workerRun.getLocalRun

Description

runObj = workerRun.getLocalRun creates the local Simulink.sdi.Run object runObj for the Simulink.sdi.WorkerRun object workerRun. Use getLocalRun in the client MATLAB to access the Simulink.sdi.WorkerRun data.

Output Arguments

runObj — Local Simulink.sdi.Run object
Simulink.sdi.Run object

Local Simulink.sdi.Run object.

Examples

Access Data from a Parallel Simulation

This example executes parallel simulations of the model slexAircraftExample with different input filter time constants and shows several ways to access the data using the Simulation Data Inspector programmatic interface.
Setup

Start by ensuring the Simulation Data Inspector is empty and Parallel Computing Toolbox support is configured to import runs created on local workers automatically. Then, create a vector of filter parameter values to use in each simulation.

% Make sure the Simulation Data Inspector is empty, and PCT support is enabled.
Simulink.sdi.clear
Simulink.sdi.enablePCTSupport('local')

% Define Ts values
Ts_vals = [0.01, 0.02, 0.05, 0.1, 0.2, 0.5, 1];

Initialize Parallel Workers

Use gcp to create a pool of local workers to run parallel simulations if you don't already have one. In an spmd code block, load the slexAircraftExample model and select signals to log. To avoid data concurrency issues using sim in parfor, create a temporary directory for each worker to use during simulations.

p = gcp;
Starting parallel pool (parpool) using the 'local' profile ...
connected to 4 workers.

spmd

% Load system and select signals to log
load_system('slexAircraftExample')
Simulink.sdi.markSignalForStreaming('slexAircraftExample/Pilot', 1, 'on')
Simulink.sdi.markSignalForStreaming('slexAircraftExample/Aircraft Dynamics Model',)

% Create temporary directory on each worker
workDir = pwd;
addpath(workDir)
tempDir = tempname;
mkdir(tempDir)
cd(tempDir)
end
Run Parallel Simulations

Use `parfor` to run the seven simulations in parallel. Select the value for `Ts` for each simulation, and modify the value of `Ts` in the model workspace. Then, run the simulation and build an array of `Simulink.sdi.WorkerRun` objects to access the data with the Simulation Data Inspector. After the `parfor` loop, use another `spmd` segment to remove the temporary directories from the workers.

```matlab
parfor index = 1:7
    % Select value for Ts
    Ts_val = Ts_vals(index);

    % Change the filter time constant and simulate
    modelWorkspace = get_param('slexAircraftExample','modelworkspace');
    modelWorkspace.assignin('Ts',Ts_val);
    sim('slexAircraftExample')

    % Create a worker run for each simulation
    workerRun(index) = Simulink.sdi.WorkerRun.getLatest
end

spmd

    % Remove temporary directories
    cd(workDir)
    rmdir(tempDir, 's')
    rmpath(workDir)
end

Get Dataset Objects from Parallel Simulation Output

The `getDataset` method puts the data from a `WorkerRun` into a `Dataset` object so you can easily post-process.

```matlab
ds(7) = Simulink.SimulationData.Dataset;
for a = 1:7
    ds(a) = workerRun(a).getDataset;
end
ds(1)
```
Get DatasetRef Objects from Parallel Simulation Output

For big data workflows, use the `getDatasetRef` method to reference the data associated with the `WorkerRun`.

```matlab
for b = 1:7
    datasetRef(b) = workerRun(b).getDatasetRef;
end
```

```matlab
datasetRef(1)
```

```matlab
ans =
    DatasetRef with properties:
        Name: 'Run 3: slexAircraftExample'
        Run: [1x1 Simulink.sdi.Run]
    numElements: 2
```

Process Parallel Simulation Data in the Simulation Data Inspector

You can also create local `Run` objects to analyze and visualize your data using the Simulation Data Inspector API. This example adds a tag indicating the filter time constant value for each run.

```matlab
for c = 1:7
    Runs(c) = workerRun(c).getLocalRun;
    Ts_val_str = num2str(Ts_vals(c));
    desc = strcat('Ts = ', Ts_val_str);
    Runs(c).Description = desc;
    Runs(c).Name = strcat('slexAircraftExample run Ts=', Ts_val_str);
end
```
Clean Up Worker Repositories

Clean up the files used by the workers to free up disk space for other simulations you want to run on your worker pool.

Simulink.sdi.cleanupWorkerResources

See Also

Simulink.sdi.Run | Simulink.sdi.WorkerRun

Topics

“Inspect and Compare Data Programmatically”

Introduced in R2017b
Simulink.Signal

Specify attributes of signal

Description

This class enables you to create workspace objects that you can use to assign or validate the attributes of a signal or discrete state, such as its data type, numeric type, dimensions, and so on.

You can use a signal object to:

- Assign values to signal attributes that are left unassigned (have a value of -1 or auto) by the signal source.
- Validate signal attributes whose values are explicitly assigned by the signal source. Such attributes have values other than -1 or auto. Successful validation guarantees that the signal has the attributes that you intended it to have.

You can create a Simulink.Signal object in the MATLAB workspace or in a model workspace.

Use signal objects to assign or validate signal or discrete state attributes by giving the signal or discrete state the same name as the workspace variable that references the Simulink.Signal object.

For more information about using signal objects, see “Use Simulink.Signal Objects to Specify and Control Signal Attributes” and “Data Objects”.

Creation

Create a Simulink.Signal object:

- By using the Model Data Editor. See “For Signals”.
- By using the Model Explorer. See “Create Data Objects from Built-In Data Class Package Simulink”.
• Directly from a signal properties dialog box or the Property Inspector in a model. See “Create Signal Object from Signal Properties Dialog Box”.
• By using the Simulink.Signal function, described below.

## Syntax

```matlab
signalObj = Simulink.Signal
```

## Description

`signalObj = Simulink.Signal` returns a `Simulink.Signal` object with default property values.

## Properties

For information about properties in the property dialog box of a `Simulink.Signal` object, see “Property Dialog Box”.

**CoderInfo — Specifications for generating code for signal**

`Simulink.CoderInfo` object

Information used by Simulink Coder for generating code for this signal. The value of this property is an object of `Simulink.CoderInfo` class.

For more information, see “Apply Storage Classes to Individual Signal, State, and Parameter Data Elements” (Simulink Coder) and “Choose Storage Class for Controlling Data Representation in Generated Code” (Embedded Coder).

**Complexity — Numeric complexity of signal**

`'auto'` (default) | `'real'` | `'complex'`

Character vector specifying the numeric type of this signal. Valid values are `'auto'` (determined by Simulink), `'real'`, or `'complex'`.

Data Types: `char`

**DataType — Data type of signal**

`'auto'` (default) | character vector
Character vector specifying the data type of this signal.

The default value, 'auto', specifies that Simulink should determine the data type. You can specify a built-in data type (for example, 'uint8' or 'single') or a custom data type. To specify a custom data type, use a MATLAB expression that specifies the type, (for example, the name of a Simulink.NumericType object that you create in the base workspace).

To specify a bus object as the data type for the signal object, use the 'Bus: <object_name>' syntax. See “Bus Support” for details about what you need to do if you specify a bus object as the data type.

Example: 'auto'
Example: 'int8'
Example: 'fixdt(1,16,5)'
Example: 'myAliasTypeObject'
Example: 'Enum: myEnumType'
Example: 'Bus: myBusObject'

Data Types: char

Description — Custom description of signal

'' (empty character vector) (default) | character vector

Description of this signal. This field is intended for use in documenting this signal.

This property is used by the Simulink Report Generator and for code generation.

If you have an Embedded Coder license, you can add the signal description as a comment for the variable declaration in generated code:

• Specify a storage class for the signal object other than Auto.
• On the Code Generation > Comments pane of the model Configuration Parameters dialog box, select the model configuration parameter Simulink data object descriptions. For more information, see “Simulink data object descriptions” (Simulink Coder).

Example: 'This signal represents the rotation speed of the engine.'

Data Types: char
**Dimensions — Dimensions of signal**  
-1 (default) | row vector | character vector

Scalar or vector specifying the dimensions of this signal.

Valid values are -1 (the default) specifying any dimensions, N specifying a vector signal of size N, or [M N] specifying an MxN matrix signal.

To use symbolic dimensions, specify a character vector.

Example: [1 3]  
Example: '[1 myDimParam]'  
Data Types: double | char

**DimensionsMode — Dimension mode of signal**  
'auto' (default) | 'Fixed' | 'Variable'

Dimensions mode of the signal. Valid values are:

- 'auto' — Allows variable-size and fixed-size signals.
- 'Fixed' — Allows only fixed-size signals. Does not allow variable-size signals.
- 'Variable' — Allows only variable-size signals.

For information about variable-size signals, see “Variable-Size Signal Basics”.

**Max — Maximum value of signal**  
[] (empty) (default) | real double scalar

Maximum value that this signal can have.

The default value is [] (unspecified). Specify a finite, real, double, scalar value.

**Note**  If you specify a bus object as the data type for a signal, do not set the maximum value for bus data on the signal property dialog box. Simulink ignores this setting. Instead, set the maximum values for bus elements of the bus object specified as the data type. For information on the Maximum property of a bus element, see `Simulink.BusElement`.

Simulink uses this value in the following ways:
When updating the diagram or starting a simulation, Simulink generates an error if the initial value of the signal is greater than the maximum value or if the maximum value is outside the range of the data type of the signal.

When you enable the **Simulation range checking** diagnostic, Simulink alerts you during simulation if the signal value is greater than the maximum value (see “Simulation range checking”).

Example: 5.32

Data Types: double

**Min — Minimum value of signal**

[ ] (empty) (default) | real double scalar

Minimum value that this signal can have.

The default value is [ ] (unspecified). Specify a finite, real, double, scalar value.

**Note** If you specify a bus object as the data type for a signal, do not set the minimum value for bus data on the signal property dialog box. Simulink ignores this setting. Instead, set the minimum values for bus elements of the bus object specified as the data type. For information on the Minimum property of a bus element, see Simulink.BusElement.

Simulink uses this value in the following ways:

- When updating the diagram or starting a simulation, Simulink generates an error if the signal's initial value is less than the minimum value or if the minimum value is outside the range for the data type of the signal.

- When you enable the **Simulation range checking** diagnostic, Simulink alerts you during simulation if the signal value is less than the minimum value (see “Simulation range checking”).

Example: -0.92

Data Types: double

**InitialValue — Initial value of signal or state**

'' (empty character vector) (default) | character vector

Signal or state value before a simulation takes its first time step.
You can specify any MATLAB expression, including the name of a workspace variable, that evaluates to a numeric scalar value or array.

At the command prompt or in a script, even if you use a number, specify the initial value as a character vector.

```matlab
mySigObject.InitialValue='5.3';
mySigObject.InitialValue = 'myNumericVariable';
```

To specify an initial value for a signal that uses a numeric data type other than `double`, cast the initial value to the signal data type. For example, you can specify `'single(73.3)'` to use 73.3 as the initial value for a signal of data type `single`.

If you use a bus object as the data type for the signal object, set `InitialValue` to a character vector containing either 0 or a MATLAB structure that matches the bus object. See “Bus Support” for details.

If the initial value evaluates to a MATLAB structure, then in the **Configuration Parameters** dialog box, set “Underspecified initialization detection” to *Simplified*.

If necessary, Simulink converts the initial value to ensure type, complexity, and dimension consistency with the corresponding block parameter value. If you specify an invalid value or expression, an error message appears when you update the model. Also, Simulink performs range checking of the initial value. The software alerts you when the initial value of the signal lies outside a range that corresponds to its specified minimum and maximum values and data type.

**Classic initialization mode:** In this mode, initial value settings for signal objects that represent the following signals and states override the corresponding block parameter initial values if undefined (specified as `[]`):

- Output signals of conditionally executed subsystems and Merge blocks
- Block states

**Simplified initialization mode:** In this mode, if a signal object is attached to the signal line connected to the Outport block of a conditionally executed subsystem, Simulink uses the `InitialValue` from the Simulink.signal object.

Example: `'15.23'`
Example: `'myInitParam'`

Data Types: `char`
SampleTime — Sample time of signal
-1 (default) | double scalar or array

Rate at which this signal should be updated.
See “Specify Sample Time” for details.
Example: 0.001
Example: 2
Data Types: double

Unit — Physical unit of signal value
'' (default) | valid unit

Physical unit used for expressing this signal value (for example, inches).
For more information, see “Unit Specification in Simulink Models”.
Example: 'degC'
Data Types: char

Examples

Simulink.Signal Examples
For examples that show how to use Simulink.Signal objects, see “Use Simulink.Signal Objects to Specify and Control Signal Attributes” and “Data Objects”.

See Also
AUTOSAR.Signal | Simulink.CoderInfo | Simulink.Parameter

Topics
“Determine Where to Store Variables and Objects for Simulink Models”
“Control Signal Data Types”
“Apply Storage Classes to Individual Signal, State, and Parameter Data Elements” (Simulink Coder)
“Define Data Classes”
“Signal Basics”
“Data Objects”
“Data Types Supported by Simulink”
“MPT Data Object Properties” (Embedded Coder)

Introduced before R2006a
Simulink.SimulationData.BlockPath

Fully specified Simulink block path

Description

Simulink creates block path objects when creating dataset objects for signal logging and data store logging. Simulink.SimulationData.Signal and Simulink.SimulationData.DataStoreMemory objects include block path objects.

You can create a block path that you can use with the Simulink.SimulationData.Dataset.getElement method to access a specific dataset element. If you want to create a dataset in MATLAB to use as a baseline to compare against a signal logging or data store logging dataset, then you need to create the block paths as part of that dataset.

The Simulink.SimulationData.BlockPath class is very similar to the Simulink.BlockPath class.

You do not have to have Simulink installed to use the Simulink.SimulationData.BlockPath class. However, you must have Simulink installed to use the Simulink.BlockPath class. If you have Simulink installed, consider using Simulink.BlockPath instead of Simulink.SimulationData.BlockPath, because the Simulink.BlockPath class includes a method for checking the validity of block path objects without you having to update the model diagram.

Property Summary

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SubPath on page 5-228</td>
<td>Individual component within the block specified by the block path</td>
</tr>
</tbody>
</table>
Method Summary

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BlockPath on page 5-229</td>
<td>Create a block path.</td>
</tr>
<tr>
<td>convertToCell on page 5-231</td>
<td>Convert a block path to a cell array of character vectors.</td>
</tr>
<tr>
<td>getBlock on page 5-232</td>
<td>Get a single block path in the model reference hierarchy.</td>
</tr>
<tr>
<td>getLength on page 5-233</td>
<td>Get the length of the block path.</td>
</tr>
</tbody>
</table>

Properties

SubPath

Description

Represents an individual component within the block specified by the block path.

For example, if the block path refers to a Stateflow chart, you can use SubPath to indicate the chart signals. For example:

Block Path:
'sf_car/shift_logic'

    SubPath:
      'gear_state.first'

Data Type

character vector

Access

RW
Methods

BlockPath

Purpose

Create block path

Syntax

blockpath_object = Simulink.SimulationData.BlockPath()
blockpath_object = Simulink.SimulationData.BlockPath(blockpath)
blockpath_object = Simulink.SimulationData.BlockPath(paths)
blockpath_object = Simulink.SimulationData.BlockPath(paths, subpath)

Input Arguments

blockpath

The block path object that you want to copy.

paths

A character vector or cell array of character vectors that Simulink uses to build the block path.

Specify each character vector in order, from the top model to the specific block for which you are creating a block path.

Each character vector must be a path to a block within the Simulink model. The block must be:

• A block in a single model
• A Model block (except for the last character vector, which may be a block other than a Model block)
• A block that is in a model that is referenced by a Model block that is specified in the previous character vector

subpath

A character vector that represents an individual component within a block.
**Output Arguments**

blockpath_object

The block path that you create.

**Description**

blockpath_object = Simulink.SimulationData.BlockPath() creates an empty block path.

blockpath_object = Simulink.SimulationData.BlockPath(blockpath) creates a copy of the block path of the block path object that you specify with the source_blockpath argument.

blockpath = Simulink.SimulationData.BlockPath(paths) creates a block path from the character vector or cell array of character vectors that you specify with the paths argument. Each character vector represents a path at a level of model hierarchy.

blockpath = Simulink.SimulationData.BlockPath(paths, subpath) creates a block path from the character vector or cell array of character vectors that you specify with the paths argument and creates a path for the individual component (for example, a signal) of the block.

**Example**

Create a block path object called bp1, using a cell array of character vectors representing elements of the block path.

bp1 = Simulink.SimulationData.BlockPath(...
{'sldemo_mdlref_depgraph/thermostat', ...
'sldemo_mdlref_heater/Fahrenheit to Celsius', ...
'sldemo_mdlref_F2C/Gain1'})

The resulting block path reflects the model reference hierarchy for the block path.

bp1 =

    Simulink.BlockPath
    Package: Simulink

    Block Path:
    'sldemo_mdlref_depgraph/thermostat'
    'sldemo_mdlref_heater/Fahrenheit to Celsius'
    'sldemo_mdlref_F2C/Gain1
convertToCell

Purpose
Convert block path to cell array of character vectors

Syntax
cellarray = Simulink.SimulationData.BlockPath.convertToCell()

Output Arguments

cellarray
The cell array of character vectors representing the elements of the block path.

Description

cellarray = Simulink.SimulationData.BlockPath.convertToCell() converts a block path to a cell array of character vectors.

Examples

bp1 = Simulink.SimulationData.BlockPath(...
 {'sldemo_mdlref_depgraph/thermostat', ...
 'sldemo_mdlref_heater/Fahrenheit to Celsius', ...
 'sldemo_mdlref_F2C/Gain1'})
cellarray_for_bp1 = bp1.convertToCell()

The result is a cell array representing the elements of the block path.

cellarray_for_bp1 =

   'sldemo_mdlref_depgraph/thermostat'
   'sldemo_mdlref_heater/Fahrenheit to Celsius'
   'sldemo_mdlref_F2C/Gain1'

getBlock

Purpose
Get single block path in model reference hierarchy
Syntax

block = Simulink.SimulationData.BlockPath.getBlock(index)

Input Arguments

index

The index of the block for which you want to get the block path. The index reflects the level in the model reference hierarchy. An index of 1 represents a block in the top-level model, an index of 2 represents a block in a model referenced by the block of index 1, and an index of n represents a block that the block with index n-1 references.

Output Arguments

block

The block representing the level in the model reference hierarchy specified by the index argument.

Description

blockpath = Simulink.SimulationData.BlockPath.getBlock(index) returns the block path of the block specified by the index argument.

Example

Get the block for the second level in the model reference hierarchy.

definePath = Simulink.SimulationData.BlockPath(...
    {'sldemo_mdlref_depgraph/thermostat', ...
    'sldemo_mdlref_heater/Fahrenheit to Celsius', ...
    'sldemo_mdlref_F2C/Gain1'})

blockpath = definePath.getBlock(2)

The result is the thermostat block, which is at the second level in the block path hierarchy.

blockpath =

sldemo_mdlref_heater/Fahrenheit to Celsius
**getLength**

**Purpose**

Get length of block path

**Syntax**

```matlab
length = Simulink.SimulationData.BlockPath.getLength()
```

**Output Arguments**

- `length`:
  
  The length of the block path. The length is the number of levels in the model reference hierarchy.

**Description**

`length = Simulink.SimulationData.BlockPath.getLength()` returns a numeric value that corresponds to the number of levels in the model reference hierarchy for the block path.

**Example**

Get the length of block path `bp1`.

```matlab
bp1 = Simulink.SimulationData.BlockPath(...
{'sldemo_mdlref_depgraph/thermostat', ...
'sldemo_mdlref_heater/Fahrenheit to Celsius', ...
'sldemo_mdlref_F2C/Gain1'})
length_bp1 = bp1.getLength()
```

The result reflects that the block path has three elements.

```matlab
length_bp1 =
    3
```

**See Also**

- `Simulink.BlockPath`
- `Simulink.SimulationData.Dataset`
Introduced in R2012b
Simulink.SimulationData.Dataset class

Package: Simulink.SimulationData

Create Simulink.SimulationData.Dataset object

Description

Simulink creates Simulink.SimulationData.Dataset objects to store data elements when:

- Performing signal logging, which use the Dataset format
- Logging states or outputs, if you use the default format of Dataset.
- Logging a data store

Using the Dataset format for state and output logging offers several advantages compared to Array, Structure, or Structure with time. For details, see “Format for State Information Saved Without Operating Point”.

To generate a Simulink.SimulationData.Dataset object from the root-level Inport blocks in a model, you can use the createInputDataset function. Signals in the generated dataset have the properties of the Inport blocks and the corresponding ground values at model start and stop times. You can create timeseries and timetable objects for the time and values for signals for which you want to load data for simulation. The other signals use ground values.

You can use curly braces ({{}) to streamline indexing syntax to access, set, and add elements in a dataset, instead of using get, getElement, setElement, or addElement methods. To get or set an element using curly braces, the index must be a scalar that is not greater than the number of elements in the dataset variable. To add an element, the index must be a scalar that is greater than the total number of elements in the dataset by one. The get, getElement, setElement, or addElement methods support specifying an element by name or block path, as well as by index.

For individual non-bus signal data, you can specify these types of data for Dataset elements:
• timeseries
• timetable
• matlab.io.datastore.SimulationDatastore
• double vectors or structure of double data
• timeseries

For bus signals, use a structure with a data element for each leaf signal, using one of these formats:

• A MATLAB timeseries object
• A MATLAB timetable object
• A matlab.io.datastore.SimulationDatastore object
• An empty matrix
• An array that meets one of these requirements:
  • An array with time in the first column and the remaining columns each corresponding to an input port. See “Loading Data Arrays to Root-Level Inputs”.
  • An nx1 array for a root inport that drives a function-call subsystem.
  • Another structure, with data elements for each signal that are consistent with these requirements for a structure for bus data

Variable-size signals are not supported for Dataset data values.

**Construction**

convertedDataset = Simulink.SimulationData.Dataset(
loggedDataToConvert) converts the loggedDataToConvert to a
Simulink.SimulationData.Dataset object. You can then use the
Simulink.SimulationData.Dataset.concat method to combine elements of two
Dataset objects.

constructedDataset = Simulink.SimulationData.Dataset(
variableName,'DatasetName','dsname') constructs a
Simulink.SimulationData.Dataset object, adds variable variableName, and
names the data set dsname.
Input Arguments

**logedDataToConvert — Data element to convert**
character vector

Data element to convert to a data set, specified as a character vector. You can convert elements such as:

- Array
- Structure

**Note**  Structure inputs cannot be arrays or matrices.

- Structure with time
- MATLAB time series
- Structure of MATLAB time-series elements
- `ModelDataLogs`

**variableName — Variable to add to data set**
character vector

Variable to add to data set, specified as a character vector.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'DatasetName','dsname'`

**DatasetName — Data set name**
character vector

Data set name, specified as a character vector.
Output Arguments

convertedDataset — Converted data set
Simulink.SimulationData.Dataset object

Converted data set, returned as a Simulink.SimulationData.Dataset object.

constructedDataset — Constructed data set
Simulink.SimulationData.Dataset object

Constructed data set, returned as a Simulink.SimulationData.Dataset object.

Properties

Name — Name of the data set
same as the logging variable (default) | character vector

Name of the data set, specified as a character vector or logging variable (for example, logsout for signal logging). Specify a name when you want to distinguish easily one data set from another. For example, you could reset the name when comparing multiple simulations. This property is read/write.

ds = Simulink.SimulationData.Dataset
ds.Name = 'Dataset1'

Total Elements — Total number of elements
double

Total number of elements in data set, specified as a double. This property is read only. To get this value, use the Simulink.SimulationData.Dataset.numElements method.
## Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>addElement</td>
<td>Add element to end of data set</td>
</tr>
<tr>
<td>concat</td>
<td>Concatenate dataset to another dataset</td>
</tr>
<tr>
<td>exportToPreviousRelease</td>
<td>Save a <code>Dataset</code> object to a MAT-file you can open in any release</td>
</tr>
<tr>
<td>get</td>
<td>Get element or collection of elements from dataset</td>
</tr>
<tr>
<td>getElementNames</td>
<td>Return names of all elements in dataset</td>
</tr>
<tr>
<td>find</td>
<td>Get element or collection of elements from dataset</td>
</tr>
<tr>
<td>numElements</td>
<td>Get number of elements in data set</td>
</tr>
<tr>
<td>plot</td>
<td>Plot data in the Simulation Data Inspector</td>
</tr>
<tr>
<td>setElement</td>
<td>Change data element stored at specified index</td>
</tr>
</tbody>
</table>

**Tip** To get the names of `Dataset` variables in the MAT-file, using the `Simulink.SimulationData.DatasetRef.getDatasetVariableNames` function processes faster than using the `who` or `whos` functions.

## Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects (MATLAB).

## Examples

### Concatenate Dataset `ds1` to Dataset `ds`

Convert data from two To Workspace blocks, convert to `Dataset` format, and concatenate them. `myvdp` is the `vdp` model with two To Workspace blocks with variables named `simout` and `simout1`. These blocks log data in time-series format.

```python
dml = 'myvdp';
open_system(dml);
sim(dml)
ds = Simulink.SimulationData.Dataset(simout);
```
ds1 = Simulink.SimulationData.Dataset(simout1);
dsfinal = concat(ds,ds1)

**Access, Change, and Add Dataset Elements**

Use curly brace indexing syntax to work with a `logsout` signal logging dataset that has three elements.

Get the second element of the `logsout` dataset.

`logsout{2}`

Change the name of the third element.

`logsout{3}.Name = 'thirdSignal'`

Add a fourth element.

```matlab
time = 0.1*(0:100)';
element4 = Simulink.SimulationData.Signal;
element4.Name = 'C';
element4.Values = timeseries(3*sin(time),time);
logsout{4} = element4
```

**See Also**


**Topics**

“Convert Logged Data to Dataset Format”
“Export Signal Data Using Signal Logging”
“Log Data Stores”
“Migrate Scripts That Use Legacy ModelDataLogs API”
“Load Big Data for Simulations”

**Introduced in R2011a**
Simulink.SimulationData.DatasetRef class

Package: Simulink.SimulationData

Create Simulink.SimulationData.DatasetRef object

Description

To use a reference for accessing a Simulink.SimulationData.Dataset object stored in a MAT-file, create a Simulink.SimulationData.DatasetRef object. You can use this reference to avoid running out of memory, by retrieving data signal by signal, for data that you log to persistent storage. You can stream a DatasetRef object into a root-level input port or you can use it to create a SimulationDatastore object to use for streaming. For details, see “Load Big Data for Simulations”.

For parallel simulations, for which you specify an array of Simulink.SimulationInput objects, if you are logging to file, Simulink:

- Creates Simulink.SimulationData.DatasetRef objects to access output data in the MAT-file and includes those objects in the SimulationOutput object data
- Enables the CaptureErrors argument for simulation

Construction


Input Arguments

location — MAT-file containing Simulink.SimulationData.Dataset object to reference character vector
MAT-file containing `Simulink.SimulationData.Dataset` object to reference, specified as a character vector. The character vector is a path to the MAT-file. Do not use a file name from one locale in a different locale.

**identifier — Name of variable in MAT-file**

character vector

Name of a `Simulink.SimulationData.Dataset` variable in MAT-file, specified as a character vector. When you log to persistent storage, Simulink uses the variable names specified for each kind of logging.

Suppose that you use the default variable name for signal logging (`logsout`) and default MAT-file name for persistent storage logging (`mat.out`), After you simulate the model, then to create a reference to the `Dataset` object for signal logging, at the MATLAB command line, enter:

```matlab
sigLogRef = Simulink.SimulationData.DatasetRef('out.mat','logsout');
```

**Output Arguments**

**datasetRefObj — Reference to Dataset object**

`Simulink.SimulationData.DatasetRef` object

Reference to logging dataset, returned as a `Simulink.SimulationData.DatasetRef` object.

**Properties**

**Location — MAT-file containing Simulink.SimulationData.Dataset object to reference**

character vector

MAT-file containing `Simulink.SimulationData.Dataset` object to reference, specified as a character vector. The character vector is a path to the MAT-file. Include the `.mat` extension in the file name. Do not use a file name from one locale in a different locale.

**Identifier — Name of variable in MAT-file**

character vector
Name of a Simulink.SimulationData.Dataset variable in MAT-file, specified as a character vector. When you log to persistent storage, Simulink uses the variable names specified for each kind of logging (for example, 'logsout' for signal logging data).

Methods

Use the numElements, getElement, and getElementNames methods for a Simulink.SimulationData.DatasetRef object the same way that you use those methods for a Simulink.SimulationData.Dataset object.

<table>
<thead>
<tr>
<th>Method</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>numElements</td>
<td>Get number of elements from dataset</td>
</tr>
<tr>
<td>getElementNames</td>
<td>Return names of all elements in dataset</td>
</tr>
<tr>
<td>get</td>
<td>Get element from dataset</td>
</tr>
</tbody>
</table>

The get method is an alias for the getElement method.

Note You can use curly braces to streamline indexing syntax to access elements in a dataset reference, instead of using get or getElement methods. To get an element using curly braces, the index must be a scalar that is not greater than the number of elements in the variable. The get and getElement methods support specifying an element by name or block path, as well as by index.

getAsDatastore

Get matlab.io.datastore.SimulationDatastore representation of element from a DatasetRef object

getDatasetVariableNames on page 2-1031

List names of Dataset variables in MAT-file
### Method

<table>
<thead>
<tr>
<th>Method</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>plot</td>
<td>Plot data in the Simulation Data Inspector</td>
</tr>
</tbody>
</table>

**Tip** To get the names of Dataset variables in the MAT-file, using the `Simulink.SimulationData.DatasetRef.getDatasetVariableNames` function processes faster than using the `who` or `whos` functions.

### Copy Semantics

You can copy `DatasetRef` object properties by value. However, copying the `DatasetRef` object produces a handle object. Copying the `DatasetRef` object does not copy the data in the MAT-file that the object references. For details about copy operations, see Copying Objects (MATLAB) in the MATLAB documentation.

### Examples

#### Create References to Persistent Storage Dataset Objects

This example shows how to construct and use `Simulink.SimulationData.DatasetRef` objects to access data for a model that logs to persistent storage. This simple example shows the basic steps for logging to persistent storage. This example does not represent a realistic situation for logging to persistent storage, because it shows a short simulation with small memory requirements.

Open the vdp model.

In the **Configuration Parameters > Data Import/Export** pane, select these parameters:

- **States**
- **Log Dataset data to file**

Set the **Format** parameter to **Dataset**.

Leave the other parameter settings as they are and click **Apply**.
In the model, click a signal and from the action bar, select **Enable Data Logging**.

Simulate the model.

Get a list of Dataset variable names in the `out.mat` file.

```matlab
varNames = Simulink.SimulationData.DatasetRef.getDatasetVariableNames('out.mat')
```

```plaintext
varNames =
    1x2 cell array
    'logsout'    'xout'
```

Create a reference to the logged states data that is stored in `out.mat`. The variable for the logged states data is `xout`.

```matlab
statesLogRef = Simulink.SimulationData.DatasetRef('out.mat','xout')
```

```plaintext
Simulink.SimulationData.DatasetRef
Characteristics:
    Location: out.mat (/my_files/out.mat)
    Identifier: xout

Resolved Dataset: 'xout' with 2 elements

    Name   BlockPath
    _____  _________
    1      vdp/x1
    2      vdp/x2
```

Create a reference to the signal logging data that is stored in `out.mat`. The variable for the signal logging data is `logsout`.

```matlab
sigLogRef = Simulink.SimulationData.DatasetRef('out.mat','logsout')
```

```plaintext
Simulink.SimulationData.DatasetRef
Characteristics:
    Location: out.mat (/my_files/out.mat)
    Identifier: logsout
```
Resolved Dataset: 'logsout' with 1 element

<table>
<thead>
<tr>
<th>Name</th>
<th>BlockPath</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>x1</td>
</tr>
<tr>
<td></td>
<td>vdp/x1</td>
</tr>
</tbody>
</table>

Use the `numElements` to access the number of elements in the logged states dataset.

```matlab
statesLogRef.numElements
```
ans =

2

Use the `DatasetRef` to access the first element of the signal logging dataset.

```matlab
sigLogRef{1}
```
ans =

```
Simulink.SimulationData.Signal
Package: Simulink.SimulationData

Properties:
  Name: 'x1'
  PropagatedName: ''
  BlockPath: [1x1 Simulink.SimulationData.BlockPath]
  PortType: 'outport'
  PortIndex: 1
  Values: [1x1 timeseries]
```

Methods, Superclasses

Delete the persistent storage MAT-file and try to use one of the `DatasetRef` objects.

```matlab
delete('out.mat');
statesLogRef.get(1)
```
File does not exist.
The `statesLogRef` still exists, but it is a reference to a `Dataset` object that is in a file that no longer exists.

**See Also**
- `Simulink.SimulationData.Dataset`
- `Simulink.SimulationData.DatasetRef.getDatasetVariableNames`
- `matlab.io.datastore.SimulationDatastore`

**Topics**
- “Log Data to Persistent Storage”
- “Load Big Data for Simulations”
- “Convert Logged Data to Dataset Format”

**Introduced in R2016a**
Simulink.SimulationData.DataStoreMemory

Container for data store logging information

Description

Simulink uses Simulink.SimulationData.DataStoreMemory objects to store logging information from Data Store Memory blocks during simulation. The objects contain information about the blocks that write to the data store.

Property Summary

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BlockPath on page 5-723</td>
<td>Location of Data Store Memory block for the logged data store</td>
</tr>
<tr>
<td>DSMWriterBlockPaths on page 5-724</td>
<td>Location of Data Store Write blocks that write to the data store</td>
</tr>
<tr>
<td>DSMWriters on page 5-724</td>
<td>Data Store Write blocks for each signal value</td>
</tr>
<tr>
<td>Name on page 5-724</td>
<td>Name of the data store dataset</td>
</tr>
<tr>
<td>Scope on page 5-725</td>
<td>Scope of the data store: 'local' or 'global'</td>
</tr>
<tr>
<td>Values on page 5-725</td>
<td>Time and data that were logged</td>
</tr>
</tbody>
</table>

Properties

BlockPath

Description

Location of Data Store Memory block for the logged data store.

Data Type

character vector
Access
RW

**DSMWriterBlockPaths**

**Description**

Location of blocks that write to the data store. Each element of the array contains the full block path of one writer block.

**Data Type**

Vector of `Simulink.SimulationData.BlockPath` objects

**Access**

R0

**DSM Writers**

**Description**

The number of writes in the data store.

The nth element of `DSM Writers` contains the index of the element in `DSMWriterBlockPaths` that contains the block path of the writer that performed the nth write to `Values`.

**Data Type**

Integer vector

**Access**

R0

**Name**

**Description**

Name of the data store dataset
**Data Type**
character vector

**Access**
R0

**Scope**

**Description**
Scope of the data store: 'local' or 'global'

**Data Type**
character vector

**Access**
RW

**Values**

**Description**
Time and data that were logged

**Data Type**
MATLAB timeseries

**Access**
RW

**See Also**
Data Store Memory | Data Store Write | Simulink.SimulationData.Dataset | plot
Topics
“Log Data Stores”
Simulink.SimulationData.LoggingInfo

Signal logging override settings

Description

This object specifies a set of signal logging override settings.

Use a Simulink.SimulationData.LoggingInfo object to specify the signal logging override settings for a signal. You can use this object for the LoggingInfo property of a Simulink.SimulationData.SignalLoggingInfo object.

Property Summary

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DataLogging on page 5-728</td>
<td>Signal logging mode.</td>
</tr>
<tr>
<td>NameMode on page 5-728</td>
<td>Source of signal logging name.</td>
</tr>
<tr>
<td>LoggingName on page 5-729</td>
<td>Custom signal logging name.</td>
</tr>
<tr>
<td>DecimateData on page 5-729</td>
<td>Use subset of sample points.</td>
</tr>
<tr>
<td>Decimation on page 5-729</td>
<td>Decimation value (n): Simulink logs every nth data point.</td>
</tr>
<tr>
<td>LimitDataPoints on page 5-730</td>
<td>Limit number of data points to log.</td>
</tr>
<tr>
<td>MaxPoints on page 5-730</td>
<td>Maximum number of data points to log (N). The set of logged data points is the last N data points generated by the simulation.</td>
</tr>
</tbody>
</table>
Method Summary

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LoggingInfo on page 5-731</td>
<td>Create a set of signal logging override settings for a signal.</td>
</tr>
</tbody>
</table>

Properties

DataLogging

Description
Signal logging mode.

Indicates whether logging is enabled for this signal.

Data Type
logical value — \{true\} | false

Access
RW

NameMode

Description
Source of signal logging name.

Indicates whether the signal logging name is a custom name ('true') or whether the signal logging name is the same as the signal name ('false').

Data Type
logical value — true | \{false\}
**Access**

RW

**LoggingName**

**Description**

Custom signal logging name

The custom signal logging name to use for this signal, if the NameMode property is true.

**Data Type**

character vector

**Access**

RW

**DecimateData**

**Description**

Log a subset of sample points, selecting data points at a specified interval. The first sample point is always logged.

**Data Type**

logical value — true | {false}

**Access**

RW

**Decimation**

**Description**

Decimation value (n). If the DecimateData property is true, then Simulink logs every nth data point.
**Data Type**

positive integer

**Access**

RW

**LimitDataPoints**

**Description**

Limit the number of data points to log.

**Data Type**

logical value — true | {false}

**Access**

RW

**MaxPoints**

**Description**

Maximum number of data points to log (N). If the LimitDataPoints property is true, then the set of logged data points includes the last N data points generated by the simulation.

**Data Type**

positive integer

**Access**

RW
Methods

LoggingInfo

Purpose
Create a Simulink.SimulationData.LoggingInfo object.

Syntax
logging_info_object = Simulink.SimulationData.LoggingInfo()
logging_info_object = Simulink.SimulationData.LoggingInfo(object)

Input Arguments
object  
A signal logging override settings object whose property values the constructor uses for the new Simulink.SimulationData.LoggingInfo object. The signal logging override object that you specify must be one of the following types of objects:

• Simulink.SimulationData.LoggingInfo object
• Simulink.LoggingInfo object

Output Arguments
logging_info_object
A Simulink.SimulationData.LoggingInfo object.

Description
logging_info_object = Simulink.SimulationData.LoggingInfo() creates a Simulink.SimulationData.LoggingInfo object that has default property values.

logging_info_object = Simulink.SimulationData.LoggingInfo(object) creates a Simulink.SimulationData.LoggingInfo object that copies the property values from the signal logging override object that you specify with the object argument.

Examples
The following example creates a Simulink.SimulationData.LoggingInfo object with default settings, changes the DecimateData and Decimation properties, and uses
the object for the `LoggingInfo` property of a `Simulink.SimulationData.SignalLoggingInfo` object `mi`.

```matlab
open_system(docpath(fullfile(docroot, 'toolbox', 'simulink', 'examples', 'ex_mdlref_counter_bus')));
open_system(docpath(fullfile(docroot, 'toolbox', 'simulink', 'examples', 'ex_bus_logging')));
log_info = Simulink.SimulationData.LoggingInfo();
log_info.DecimateData = true;
log_info.Decimation = 2;
mi = Simulink.SimulationData.SignalLoggingInfo('ex_bus_logging');
mi.LoggingInfo = log_info
```

```matlab
Simulink.SimulationData.SignalLoggingInfo
Package: Simulink.SimulationData

BlockPath:
    'ex_bus_logging'

OutputPortIndex: 1

LoggingInfo:
    DataLogging: 1
    NameMode: 0
    LoggingName: ''
    DecimateData: 1
    Decimation: 2
    LimitDataPoints: 0
    MaxPoints: 5000
```

**See Also**

| Simulink.ModelDataLogs | Simulink.SimulationData.DataStoreMemory |
| Simulink.SimulationData.ModelLoggingInfo |
| Simulink.SimulationData.Signal |
| Simulink.SimulationData.SignalLoggingInfo |

**Topics**

“Override Signal Logging Settings from MATLAB”
“Migrate Scripts That Use Legacy ModelDataLogs API”
“Export Signal Data Using Signal Logging”
“Log Data Stores”
Introduced in R2012b
Simulink.SimulationData.ModelLoggingInfo

Signal logging override settings for a model

Description

This class is a collection of Simulink.SimulationData.SignalLoggingInfo objects that specify all signal logging override settings for a model.

Use methods and properties of this class to:

- Turn off logging for a signal or a Model block.
- Change logging settings for any signals that are marked for logging within a model.

You can control whether a top model and referenced models override signal logging settings or use the signal logging settings specified by the model. Use the LoggingMode and LogAsSpecifiedByModels properties to control which logging settings to apply.

<table>
<thead>
<tr>
<th>Logging Mode for Models</th>
<th>Property Settings</th>
</tr>
</thead>
<tbody>
<tr>
<td>For top model and all referenced models, use logging settings specified in the model.</td>
<td>Set LoggingMode to LogAllAsSpecifiedInModel.</td>
</tr>
<tr>
<td>For top model and all referenced models, use override signal logging settings.</td>
<td>Set LoggingMode to OverrideSignals.</td>
</tr>
<tr>
<td>For top model and referenced models, use a mix of override signal logging settings and the signal logging settings specified in the model.</td>
<td>Set LoggingMode to OverrideSignals.</td>
</tr>
<tr>
<td></td>
<td>Include models you want to ignore override signal logging settings in the LogAsSpecifiedByModels cell array.</td>
</tr>
</tbody>
</table>

For more information and examples, see “Override Signal Logging Settings from MATLAB”. 
Property Summary

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LoggingMode on page 5-735</td>
<td>Signal logging override status</td>
</tr>
<tr>
<td>LogAsSpecifiedByModels on page 5-736</td>
<td>Source of signal logging settings for the top model or a top-level Model block</td>
</tr>
<tr>
<td>Signals on page 5-737</td>
<td>All signals that have signal override settings</td>
</tr>
</tbody>
</table>

Method Summary

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>findSignal on page 5-742</td>
<td>Find signals within the Signals vector, using block path and output port index.</td>
</tr>
<tr>
<td>verifySignalAndModelPaths on page 5-746</td>
<td>Verify signal and model paths for the model signal logging override object.</td>
</tr>
<tr>
<td>getLogAsSpecifiedInModel on page 5-744</td>
<td>Determine whether the model logs signals as specified in the model or uses override settings.</td>
</tr>
<tr>
<td>setLogAsSpecifiedInModel on page 5-745</td>
<td>Set the logging mode for the top model or a top-level Model block.</td>
</tr>
<tr>
<td>createFromModel on page 5-737</td>
<td>Create and populate a model signal logging override object with all logged signals in the model hierarchy.</td>
</tr>
<tr>
<td>ModelLoggingInfo on page 5-740</td>
<td>Set signals to log or override logging settings.</td>
</tr>
</tbody>
</table>

Properties

LoggingMode

Description

Signal logging override status. Values are:

- **OverrideSignals** — (Default) Uses the logging settings for signals, as specified in the Signals property. For models where `getLogAsSpecifiedInModel` is:
• true — Logs all signals, as specified in the model.
• false — Logs only the signals specified in the Signals property.
• LogAllAsSpecifiedInModel — Logs signals in the top model and all referenced models, as specified in the model. Simulink honors the signal logging indicators (blue antennae) and ignores the Signals property.

To change the logging mode for the top model or for a given referenced model, use the setLogAsSpecifiedInModel method.

Data Type
character array

Access
RW

LogAsSpecifiedByModels

Description
When LoggingMode is set to 'OverrideSignals', the LogAsSpecifiedByModels cell array specifies the top models and top-level Model blocks that ignore the 'OverrideSignals' setting and log signals as specified in the models or Model blocks.

• For the top model and top-level Model blocks that the cell array includes, Simulink ignores the Signals property overrides.
• For a model or Model block that the cell array does not include, Simulink uses the Signals property to determine which signals to log.

When LoggingMode is set to 'LogAllAsSpecifiedInModel', Simulink ignores the LogAsSpecifiedByModels property.

Use the getLogAsSpecifiedInModel method to determine whether the top model or top-level Model block logs signals as specified in the model (default logging), and use setLogAsSpecifiedInModel to turn default logging on and off.

Data Type
cell array — For the top model, specify the model name. For Model blocks, specify the block path.
Access
RW

Signals

Description
Vector of Simulink.SimulationData.SignalLoggingInfo objects for all signals with signal logging override settings.

Data Type
vector of Simulink.SimulationData.SignalLoggingInfo objects

Access
RW

Methods

createFromModel

Purpose
Create a Simulink.SimulationData.ModelLoggingInfo object for a top model, with override settings for each logged signal in the model.

Syntax
model_logging_info_object = ...
Simulink.SimulationData.ModelLoggingInfo.createFromModel(...model,options)

Input Arguments
model
   Name of the top model for which to create a Simulink.SimulationData.ModelLoggingInfo object.
options
You can use any combination of the following option name and value pairs to control the kinds of systems from which to include logged signals.

- **FollowLinks**
  - on — (Default) Include logged signals from inside of libraries.
  - off — Skip all libraries.

- **LookUnderMasks**
  - all — (Default) Include logged signals from all masked subsystems.
  - none — Skip all masked subsystems.
  - graphical — Include logged signals from masked subsystems that do not have a workspace or dialog box.
  - functional — Include logged signals from masked subsystems that do not have a dialog box.

- **Variants**
  - ActiveVariants — (Default) Include logged signals from only active subsystem and model variants.
  - AllVariants — Include logged signals from all subsystem and model variants.

- **RefModels**
  - on — (Default) Include logged signals from referenced models.
  - off — Skip all referenced models.

If you select more than one option, then the created `Simulink.SimulationData.ModelLoggingInfo` object includes signals that fit the combinations (the “AND“) of the specified options. For example, if you set `FollowLinks` to on and set `RefModels` to off, then the model signal logging override object does not include signals from library links that exist inside of referenced models.

**Output Arguments**

`model_logging_override_object`

`Simulink.SimulationData.ModelLoggingInfo` object for the top model.
**Description**

\[
\text{model\_logging\_info\_object} = \text{Simulink.SimulationData.ModelLoggingInfo.createFromModel(model)}
\]
creates a Simulink.SimulationData.ModelLoggingInfo object for the model that includes logged signals for the following kinds of systems:

- Libraries
- Masked subsystems
- Referenced models
- Active variants

\[
\text{model\_logging\_override\_object} = \text{Simulink.SimulationData.ModelLoggingInfo.createFromModel(model, options)}
\]
creates a Simulink.SimulationData.ModelLoggingInfo object for the model. The included logged signals reflect the options settings for the following kinds of systems:

- Libraries
- Masked subsystems
- Referenced models
- Variants

**Examples**

The following example creates a model logging override object for the `sldemo_mdlref_bus` model and automatically adds each logged signal in the model to that object:

\[
\text{mi} = \text{Simulink.SimulationData.ModelLoggingInfo.createFromModel(... 'sldemo_mdlref_bus')}
\]

\[
\text{mi} =
\]

ModelLoggingInfo with properties:

- Model: 'sldemo_mdlref_bus'
- LoggingMode: 'OverrideSignals'
- LogAsSpecifiedByModels: {}
- Signals: [1x3 Simulink.SimulationData.SignalLoggingInfo]

To apply the model override object settings, use:

\[
\text{set\_param(sldemo_mdlref\_bus,'DataLoggingOverride',mi)};
\]
You can use the options for the `createFromModel` method to specify how the method should handle model components like variants and model references. For example, use the `Variants` option to create a `model_logging_override` object that includes logged signals in all variants of the `sldemo_variant_subsystems` model.

By default, the `sldemo_variant_subsystems` model does not log any signals. Start by configuring the output signals from the `Linear Controller` and `Nonlinear Controller` subsystems for logging.

```matlab
% Open the sldemo_variant_subsystems model sldemo_variant_subsystems;
% Mark the output of the Linear Controller subsystem for logging
ph = get_param('sldemo_variant_subsystems/Controller/Linear Controller','PortHandles');
set_param(ph.Outport(1),'DataLogging','on');
% Mark the output of the Nonlinear Controller subsystem for logging
ph1 = get_param('sldemo_variant_subsystems/Controller/Nonlinear Controller','PortHandles');
set_param(ph1.Outport(1),'DataLogging','on');
```

Then, use the `createFromModel` method to create a `model_logging_override` object that includes signals logged in all variant subsystems of the `sldemo_variant_subsystems` model.

```matlab
% Create a model_logging_override object for the model including all variants
mi = Simulink.SimulationData.ModelLoggingInfo.createFromModel(...
    'sldemo_variant_subsystems', 'Variants', 'AllVariants')
```

```matlab
mi =

ModelLoggingInfo with properties:
    Model: 'sldemo_variant_subsystems'
    LoggingMode: 'OverrideSignals'
    LogAsSpecifiedByModels: {}
    Signals: [1x2 Simulink.SimulationData.SignalLoggingInfo]
```

**ModelLoggingInfo**

**Purpose**

Specify signals to log or override logging settings.
Syntax

model_logging_override_object = ....
Simulink.SimulationData.ModelLoggingInfo(model)

Input Arguments

model

Name of the top model for which to create a
Simulink.SimulationData.ModelLoggingInfo object

Output Arguments

model_logging_override_object

Simulink.SimulationData.ModelLoggingInfo object created for the specified
top model.

Description

model_logging_override_object=
Simulink.SimulationData.ModelLoggingInfo(model) creates a
Simulink.SimulationData.ModelLoggingInfo object for the specified top model.

If you use the Simulink.SimulationData.ModelLoggingInfo constructor, specify a
Simulink.SimulationData.SignalLoggingInfo object for each logged signal for
which you want to override logging settings.

To check that you have specified valid signal logging override settings for a model, use
the verifySignalAndModelPaths method with the
Simulink.SimulationData.ModelLoggingInfo object for the model.

Examples

The following example shows how to log all signals as specified in the top model and all
referenced models.

mi = Simulink.SimulationData.ModelLoggingInfo('sldemo_mdlref_bus');
mi.LoggingMode = 'LogAllAsSpecifiedInModel'

mi =
ModelLoggingInfo with properties:
    Model: 'sldemo_mdlref_bus'
    LoggingMode: 'LogAllAsSpecifiedInModel'
To apply the model override object settings, use:

```matlab
set_param(sldemo_mdlref_bus, 'DataLoggingOverride', mi);
```

The following example shows how to log only signals in the top model:

```matlab
mi = ...;
Simulink.SimulationData.ModelLoggingInfo('sldemo_mdlref_bus');
mi.LoggingMode = 'OverrideSignals';
mi = mi.setLogAsSpecifiedInModel('sldemo_mdlref_bus', true);
set_param('sldemo_mdlref_bus', 'DataLoggingOverride', mi);
```

## findSignal

### Purpose

Find signals within the Signals vector, using a block path and optionally an output port index.

### Syntax

```matlab
signal_indices = ...
    model_logging_override_object.findSignal(block_path)
signal_indices = ...
    model_logging_override_object.findSignal(...
        block_path, port_index)
```

### Input Arguments

- **block_path**
  - Source block to search. The block path must be one of the following:
    - Character vector
    - Cell array of character vectors
    - Simulink.BlockPath object

- **port_index**
  - Index of the output port to search. Specify a scalar greater than, or equal to, 1.
Output Arguments

signal_indices

Vector of numeric indices into the signals vector of the

Description

signal_indices = model_logging_override_object.findSignal(
block_path) finds the indices of the signals for the block path that you specify.

To find a single instance of a signal within a referenced model, use a
Simulink.BlockPath object or a cell array with a full path.

To find all instances of a signal within a referenced model, use a character vector with the
relative path of the signal within the referenced model.

To find a logged chart signal within a Stateflow chart, use a Simulink.BlockPath
object and set the SubPath property to the name of the Stateflow chart signal.

signal_indices = model_logging_override_object.findSignal(
block_path, port_index) finds the indices of the output signal for the port that you
specify, for the block path that you specify.

Do not use the port_index argument for Stateflow chart signals.

Examples

To find a signal that is not in a Stateflow chart and that does not appear in multiple
instances of a referenced model:

open_system(docpath(fullfile(docroot,'toolbox','simulink','examples','ex_bus_logging')))
% Open the referenced model
ex_mdlref_counter_bus
mi = Simulink.SimulationData.ModelLoggingInfo.createFromModel(...
    'ex_bus_logging');
% Click the COUNTERBUSCreator block that is the source of
% the logged COUNTERBUS signal
signal_index = mi.findSignal(gca)

signal_index =

    1
To find a signal in a specific instance of a referenced model that is not in a Stateflow chart, use the following approach:

```matlab
signal_index = mi.findSignal({'ex_bus_logging/CounterA', ...
'ex_mdlref_counter_bus/Bus Creator'})
```

```matlab
signal_index =
    4
```

For an example that uses the `findSignal` method with a Stateflow chart, see “Override Logging Properties with the Command-Line API” (Stateflow).

**getLogAsSpecifiedInModel**

**Purpose**

Determine whether the model logs as specified in the model or uses override settings.

**Syntax**

```matlab
logging_mode = ...
getLogAsSpecifiedInModel(model_logging_override_object, path)
```

**Input Arguments**

- `model_logging_override_object`:

- `path`:
  The path is a character vector that specifies one of the following:
  - Name of the top model
  - Block path of a Model block in the top model

**Output Arguments**

- `logging_mode`:
  The `logging_mode` is:
  - `true`, if the model specified by `path` is logged as specified in the model.
  - `false`, if the model specified by `path` is logged using the override settings specified in the `Signals` property.
Description

logging_mode =
model_logging_override_object.getLogAsSpecifiedInModel(path) returns:

• true, if the model specified by path is logged as specified in the model.
• false, if the model specified by path is logged using the override settings specified in the Signals property.

Examples

In the following example, the Simulink.SimulationData.ModelLoggingInfo object mi uses the override settings specified in its Signals property.

```matlab
mi = Simulink.SimulationData.ModelLoggingInfo('sldemo_mdlref_bus');
logging_mode = getLogAsSpecifiedInModel(mi, 'sldemo_mdlref_bus')
```

```matlab/logging_mode =
  0
```

setLogAsSpecifiedInModel

Purpose

Set logging mode for top model or top-level Model block

Syntax

setLogAsSpecifiedInModel(override_object, path)

Input Arguments

override_object


path

Character vector that specifies one of the following:

• Name of the top model
• Block path of a Model block in the top model
value

Logging mode:

- true, if the model specified by path is logged as specified in the model
- false, if the model specified by path is logged using the override settings
  specified in the Signals property.

Description

setLogAsSpecifiedInModel(override_object, path, value) sets the
LoggingMode property for a top model or a Model block in the top model.

Example

The following example shows how to log only signals in the top model, using the logging
settings specified in that model:

```matlab
sldemo_mdlref_bus;
mi = Simulink.SimulationData.ModelLoggingInfo('sldemo_mdlref_bus');
mi.LoggingMode = 'OverrideSignals';
mi = setLogAsSpecifiedInModel(mi, 'sldemo_mdlref_bus', true);
set_param('sldemo_mdlref_bus', 'DataLoggingOverride', mi);
```

verifySignalAndModelPaths

Purpose


Syntax

```matlab
verified_object = verifySignalAndModelPaths...
    (model_logging_override_object, action)
```

Input Arguments

- model_logging_override_object
  The Simulink.SimulationData.ModelLoggingInfo object to verify. This
  argument is required.

- action
  The action that the function performs if verification fails. This argument is optional.
  Specify one of the following values:
- **error** — (default) Throw an error when verification fails
- **warnAndRemove** — Issue a warning when verification fails and update the Simulink.SimulationData.ModelLoggingInfo object.
- **remove** — Silently update the Simulink.SimulationData.ModelLoggingInfo object.

**Output Arguments**

**verified_object**
If the method detects no invalid paths, it returns the validated object. For example:

```markdown
verified_object = 
Simulink.SimulationData.ModelLoggingInfo
Package: Simulink.SimulationData

Properties:
  Model: 'logging_top'
  LoggingMode: 'OverrideSignals'
  LogAsSpecifiedByModels: {}
  Signals: [1x11 Simulink.SimulationData.SignalLoggingInfo]
```

If the method detects an invalid path, it performs the action specified by the `action` argument. By default, it issues an error message.

**Description**

```markdown
verified_object = verifySignalAndModelPaths(
  model_logging_override_object, action)
```

For a `Simulink.SimulationData.ModelLoggingInfo` object, verify that:

- All character vectors in the `LogAsSpecifiedByModels` property are either the name of the top model or the block path of a Model block in the top model.
- The block paths for signals in the `Signals` property refer to valid blocks within the hierarchy of the top model.
- The `OutputPortIndex` property for all signals in the `Signals` property are valid for the given block.
- All signals in the `Signals` property refer to logged signals.

The `action` argument specifies what action the method performs. By default, the method returns an error if it detects an invalid path.
If you use the Simulink.SimulationData.ModelLoggingInfo constructor and specify a Simulink.SimulationData.SignalLoggingInfo object for each signal, then consider using the verifySignalAndModelPaths method to verify that your object definitions are valid.

Example

The following example shows how to validate the signal and block paths in a Simulink.SimulationData.ModelLoggingInfo object. Because the action argument is warnAndRemove, if the validation fails, the verifySignalAndModelPaths method issues a warning and updates the Simulink.SimulationData.ModelLoggingInfo object.

```matlab
mi = Simulink.SimulationData.ModelLoggingInfo('sldemo_mdlref_bus');
verified_object = verifySignalAndModelPaths(mi, 'warnAndRemove')
```

See Also


Topics

“Override Signal Logging Settings from MATLAB”
“Migrate Scripts That Use Legacy ModelDataLogs API”
“Export Signal Data Using Signal Logging”
“Log Data Stores”

Introduced in R2012b
Simulink.SimulationData.Parameter class

**Package:** Simulink.SimulationData

Stores logged parameter data and metadata

**Description**

The `Simulink.SimulationData.Parameter` object stores data and metadata for logged block parameters. Tunable parameters connected to Dashboard blocks are logged to the Simulation Data Inspector during simulation. To access logged parameter data, you can export the simulation run from the Simulation Data Inspector using the UI or the `Simulink.sdi.exportRun` function. For more information about exporting simulation runs with the Simulation Data Inspector UI, see “Export Data from the Simulation Data Inspector”.

**Construction**

```matlab
dataset = Simulink.sdi.exportRun(runID)
```

returns a `Simulink.SimulationData.Parameter` object as an element in `dataset` when the run corresponding to `runID` contains logged parameter data.

**Input Arguments**

- `runID` — Run ID for a run with logged parameter data
  
  integer

  Run ID for the run containing logged parameter data. Run IDs are assigned by the Simulation Data Inspector. You can get the run ID for a simulation run using the `Simulink.sdi.getAllRunIDs` or `Simulink.sdi.getRunIDByIndex` function.

**Output Arguments**

- `dataset` — Dataset containing the run data
  
  `Simulink.SimulationData.Dataset`
Simulink.SimulationData.Dataset object containing the run data and metadata. When the run contains logged parameter data, the dataset contains a Simulink.SimulationData.Parameter object as an element for each logged parameter. The Simulink.SimulationData.Parameter element takes the name of the logged parameter. You can access a Simulink.SimulationData.Parameter object using `get`.

**Properties**

**Name — Parameter name in Dashboard label**
character vector

Parameter name as it appears in the label for the Dashboard block.
Example: 'Mu:Gain'

**BlockPath — Path of the block associated with the parameter**
Simulink.SimulationData.BlockPath

Path to the block the parameter or variable corresponds to, returned as a Simulink.SimulationData.BlockPath object.
Example: vdp/Mu

**ParameterName — Parameter name**
character vector

Name of the logged parameter as it appears in the block dialog box. For variables, the ParameterName property is empty.
Example: 'Gain'

**VariableName — Variable name**
character vector

Name of the logged variable. For parameters, the VariableName property is empty.
Example: 'Zw'

**Values — Timeseries of parameter values**
timeseries
timeseries of parameter values. For logged variables, the timeseries name is the variable name. For logged parameters, the timeseries name is empty.

**Methods**

`plot`  Plot simulation output data in the Simulation Data Inspector

**Copy Semantics**

Value. To learn how value classes affect copy operations, see Copying Objects (MATLAB).

**Examples**

**Access Logged Parameter Data**

This example shows how to access logged parameter data. Parameter data automatically logs to the Simulation Data Inspector when you connect a Dashboard block to a block parameter. Parameter data does not export to the workspace with other simulation data at the end of simulation. You can access the logged parameter data by exporting the run containing it from the Simulation Data Inspector.

**Log Parameter Data**

Run a simulation of the model `ex_vdp_param`, a modified version of the `vdp` model with an Edit block connected to the gain parameter of the Mu block. The parameter data logs with the signal data for signals marked for logging.

```matlab
sim('ex_vdp_param');
```

**Export Run**

Use the Simulation Data Inspector programmatic interface to get the run ID for the `ex_vdp_param` simulation, and export the run.

```matlab
index = Simulink.sdi.getRunCount;
runID = Simulink.sdi.getRunIDByIndex(index);
dataset = Simulink.sdi.exportRun(runID);
```
Access Parameter Data

Use the `get` method to access the `Simulink.SimulationData.Parameter` object for the logged parameter data. The `Values` property contains the timeseries data for the parameter.

```matlab
muGain = dataset.get('Mu:Gain')
```

```
muGain =
    Simulink.SimulationData.Parameter
    Package: Simulink.SimulationData

    Properties:
        Name: 'Mu:Gain'
        BlockPath: [1x1 Simulink.SimulationData.BlockPath]
        Values: [1x1 timeseries]
```

See Also

`Simulink.SimulationData.BlockPath` | `Simulink.SimulationData.Dataset` | `get`

Topics

“Tune and Visualize Your Model with Dashboard Blocks”
“View Data with the Simulation Data Inspector”

Introduced in R2018a
Simulink.SimulationData.SignalLoggingInfo

Signal logging override settings for signal

Description

This object contains the signal override signal logging settings for a single logged signal.

Property Summary

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BlockPath on page 5-754</td>
<td>Simulink.BlockPath of source block of a signal to log.</td>
</tr>
<tr>
<td>OutputPortIndex on page 5-754</td>
<td>Index of an output port to log.</td>
</tr>
<tr>
<td>LoggingInfo on page 5-755</td>
<td>Simulink.SimulationData.LoggingInfo object containing all logging override settings for a signal.</td>
</tr>
</tbody>
</table>

Method Summary

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SignalLoggingInfo on page 5-755</td>
<td>Create a signal logging override object for a signal.</td>
</tr>
</tbody>
</table>
Properties

BlockPath

Description

Simulink.BlockPath of source block of signal to log. The block path represents the full model reference hierarchy.

To specify a specific instance of a signal, use an absolute path, reflecting the model reference hierarchy, starting at the top model. For example:

```matlab
sig_log_info = Simulink.SimulationData.SignalLoggingInfo(...
    {'sldemo_mdlref_bus/CounterA', ...
    'sldemo_mdlref_counter_bus/Bus Creator'})
```

Data Type

Simulink.BlockPath

Access

RW

OutputPortIndex

Description

Index of the output port to log. The index is a 1-based numeric value.

Data Type

nonzero integer

Access

RW
**LoggingInfo**

**Description**

`Simulink.SimulationData.LoggingInfo` object containing logging override settings for a signal. The logging settings specify whether signal logging is overridden for this signal. The logging settings also can specify a logging name, a decimation factor, and a maximum number of data points.

**Data Type**

`Simulink.SimulationData.LoggingInfo` object

**Access**

RW

**Methods**

**SignalLoggingInfo**

**Purpose**


**Syntax**

```matlab
signal_logging_info_object = ...
    Simulink.SimulationData.SignalLoggingInfo()
signal_loggingInfo_object = ...
    Simulink.SimulationData.SignalLoggingInfo(path)
signalLoggingInfo_object = ...
    Simulink.SimulationData.SignalLoggingInfo(path,index)
```

**Input Arguments**

`path`

The block path of the source block for which the signal logging override settings apply. If you use this argument without also using the `port` argument, then Simulink sets the output port index to 1.
index
Output port index to which the signal logging override settings apply.

Output Arguments

signal_logging_object
Simulink.SimulationData.SignalLoggingInfo object that represents the override settings of a signal.

Description


signal_logging_override_object = Simulink.SimulationData.SignalLoggingInfo(path) creates a Simulink.SimulationData.LoggingInfo object, using the specified block path, and sets the output port index to 1.

signal_logging_override_object = Simulink.SimulationData.SignalLoggingInfo(path, port) creates a Simulink.SimulationData.LoggingInfo object that contains default logging settings for the specified block path and output port index.

Examples

The following example creates a Simulink.SimulationData.SignalLoggingInfo object for the first output port of the Bus Creator block in the sldemo_mdlref_bus model.

```matlab
sldemo_mdlref_bus;
mi = Simulink.SimulationData.ModelLoggingInfo('sldemo_mdlref_bus');
mi.LoggingMode = 'OverrideSignals';
```

The output is:

```matlab
mi =
```
Data.ModelLoggingInfo with properties:

  Model: 'sldemo_mdlref_bus'
  LoggingMode: 'OverrideSignals'
  LogAsSpecifiedByModels: {}
  Signals: [1x1 Simulink.SimulationData.SignalLoggingInfo]

Methods

See Also

Topics
“Override Signal Logging Settings from MATLAB”
“Migrate Scripts That Use Legacy ModelDataLogs API”
“Export Signal Data Using Signal Logging”
“Log Data Stores”

Introduced in R2012b
Simulink.SimulationData.Signal

Container for signal logging information

Description

Simulink uses Simulink.SimulationData.Signal objects to store signal logging information during simulation. The objects contain information about the source block for the signal, including the port type and index.

Property Summary

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BlockPath on page 5-758</td>
<td>Block path for the source block for the signal</td>
</tr>
<tr>
<td>Name on page 5-759</td>
<td>Name of signal element to use for name-based access</td>
</tr>
<tr>
<td>PropagatedName on page 5-759</td>
<td>Propagated signal name, if any</td>
</tr>
<tr>
<td>PortIndex on page 5-760</td>
<td>Numeric index of port that was logged</td>
</tr>
<tr>
<td>PortType on page 5-760</td>
<td>Type of port that was logged: for signal logging, the port type is 'outport'</td>
</tr>
<tr>
<td>Values on page 5-760</td>
<td>Time and data that were logged</td>
</tr>
</tbody>
</table>

Properties

BlockPath

Description
Block path for the source block for the signal

Data Type
Simulink.SimulationData.BlockPath
Access
RW

Name

Description
Name of signal element to use for name-based access

Data Type
character vector

Access
RW

PropagatedName

Description
Propagated name of signal element

Signal logging and root Outport block logging data for a signal captures the propagated signal name if the logging format is Dataset and:

- For signal logging, you:
  - Mark the signal for signal logging and in the Signal Properties dialog box select Show Propagated Signals.
  - Enable Configuration Parameters > Data Import/Export > Signal logging.
- For root Outport block logging, you select Configuration Parameters > Data Import/Export > Output.

The propagated signal name does not include angle brackets (<>).

Data Type
character vector
Access
R0

**PortIndex**

**Description**
Numeric index of port that was logged

**Data Type**
scalar real integer

Access
RW

**PortType**

**Description**
Type of port that was logged: for signal logging, the port type is 'outport'

**Data Type**
character vector

Access
RW

**Values**

**Description**
Time and data that were logged.

For an example of how to use the Values property and plot logged signal data, in the sldemo_mdlref_bus example, see “Logging Signal Data.”
Data Type

- MATLAB timeseries object
- Structure of MATLAB timeseries objects (for bus signals)
- Array of structures of MATLAB timeseries objects (for array of buses signals)
- Array of MATLAB timeseries objects (for nonbus signals in a For Each subsystem)
- MATLAB timetable object
- Structure of MATLAB timetable objects (for bus signals)
- Array of structures of MATLAB timetable objects (for array of buses signals)
- Cell array of MATLAB timetable objects (for nonbus signals in a For Each subsystem)

Access

RW

See Also

Simulink.BlockPath | Simulink.SimulationData.Dataset | plot | timeseries

Topics

“View and Access Signal Logging Data”
“Loading MATLAB Timeseries Data to Root-Level Inputs”
“Load Bus Data to Root-Level Input Ports”
“Log Signals in For Each Subsystems”
Simulink.SimulationData.State class

**Package:** Simulink.SimulationData

State logging element

**Description**

Simulink uses Simulink.SimulationData.State objects to store state logging information during simulation. The objects contain state information about which block the state data is coming from and the type of state.

**Properties**

**Name** — Name of state element to use for name-based access

character vector

Name of state element to use for name-based access, specified as a character vector. If you do not specify a name, 'CSTATE' or 'DSTATE' is used, depending on whether it a continuous or discrete state.

**BlockPath** — Block path for state source block

a Simulink.SimulationData.BlockPath object

Block path for state source block, specified as a Simulink.SimulationData.BlockPath object

**Label** — Type of state

'CSTATE' | 'DSTATE'

Type of state, returned as 'CSTATE' or 'DSTATE'. Read-only property.

- 'CSTATE' - Continuous state
- 'DSTATE' - Discrete state

**Values** — State element information

single MATLAB timeseries object | a structure of MATLAB timeseries objects
State element information, specified as a single MATLAB timeseries object or as a structure of MATLAB timeseries objects.

**Methods**

plot  Plot simulation output data in the Simulation Data Inspector

**Examples**

**Final State Information in Structure with Dataset Format**

Saved final state information in Dataset format and access the state data after simulation.

Open the vdp model and specify to log final states in Dataset format. Use the default logged state variable, xFinal.

```matlab
open_system('vdp');
set_param(gcs,'SaveFinalState','on','SaveFormat','Dataset');
```
van der Pol Equation

Copyright 2004-2013 The MathWorks, Inc.
Simulate the vdp model.

sim('vdp');
View the state logging information in `xFinal`.

`xFinal`  

`xFinal =`  

`Simulink.SimulationData.Dataset 'xFinal' with 2 elements`  

<table>
<thead>
<tr>
<th>Name</th>
<th>BlockPath</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 [1x1 State]</td>
<td>vdp/x1</td>
</tr>
<tr>
<td>2 [1x1 State]</td>
<td>vdp/x2</td>
</tr>
</tbody>
</table>

- Use braces `{ }` to access, modify, or add elements using index.
Examine the first element of the state dataset.

```matlab
xFinal.get(1)
```

```matlab
ans =

Simulink.SimulationData.State
Package: Simulink.SimulationData

Properties:
    Name: ''
    BlockPath: [1x1 Simulink.SimulationData.BlockPath]
    Label: CSTATE
    Values: [1x1 timeseries]
```

**See Also**

Simulink.SimulationData.Dataset

**Topics**

“State Information”

**Introduced in R2015a**
Simulink.SimulationData.Unit class

Package: Simulink.SimulationData

Store units for simulation data

Description

Simulink creates Simulink.SimulationData.Unit objects to store unit information for signals when:

- Performing signal logging, which uses the Dataset format
- Logging root Outport blocks, if in Configurations Parameters you select the Output parameter and set Format to Dataset
- Logging to a To Workspace block or To File block, if you set the Save format block parameter to the default of Timeseries

Construction

`unitsObj = Simulink.SimulationData.Unit(unitName)` creates a Simulink.SimulationData.Unit object with the unit that you specify.

Input Arguments

- `unitName` — Name of logging data units
  character vector

Name of logging data units, specified as a character vector.

Output Arguments

- `unitObj` — Logging data units
  Simulink.SimulationData.Unit object

Logging data units, returned as a Simulink.SimulationData.Unit object.
Properties

Name — Name of the units
character vector

Name of the units, specified as a character vector.

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simulink.SimulationData.Unit.setName</td>
<td>Specify name of logging data unit</td>
</tr>
</tbody>
</table>

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects (MATLAB).

Examples

Create and Use Inches Logging Units

Create a Simulink.SimulationData.Unit object representing inches.

inchesUnit = Simulink.SimulationData.Unit('inches');

Create a MATLAB timeseries object and set its Units field to the Simulink.SimulationData.Unit object.

ts = timeseries(1:10);
ts.DataInfo.Units = inchesUnit;
ts.DataInfo.Units

ans =

    Units with properties:
Name: 'inches'

See Also
Simulink.SimulationData.Dataset

Topics
“Log Signal Data That Uses Units”
“Load Signal Data That Uses Units”
“Convert Logged Data to Dataset Format”
“Prepare Model Inputs and Outputs”

Introduced in R2016a
Simulink.SimulationInput class

Package: Simulink

Creates SimulationInput objects to make changes to a model for multiple or individual simulations

Description

The Simulink.SimulationInput object allows you to make changes to a model and run simulations with those changes. These changes are temporarily applied to the model. Through the Simulink.SimulationInput object, you can change:

- Initial state
- Model parameters
- Block parameters
- External inputs
- Variables

Through the Simulink.SimulationInput object, you can also specify MATLAB functions to run at the start and the end of each simulation by using `in.setPreSimFcn` and `in.setPostSimFcn`, respectively. Simulink.SimulationInput does not support the ability to allow model references to have their own data dictionary.

Construction

`in = Simulink.SimulationInput('modelName')` creates a SimulationInput object for a model.

Input Arguments

- `modelName` — Name of the model
character vector
Create a `Simulink.SimulationInput` object by passing the name of the model as an argument.

Example: `in = Simulink.SimulationInput('cstr')`

## Properties

**ModelName — Name of the model**
character vector

Name of the model for which the `SimulationInput` object is created.

**InitialState — Initial state**
`Simulink.op.ModelOperatingPoint` object

Initial state of the model for a simulation specified as a `Simulink.op.ModelOperatingPoint` object.

**ExternalInput — External Input**
numerical array, dataset object, timeseries, character array

External inputs added to the model for a simulation.

**BlockParameters — Block parameters**
array of `Simulink.Simulation.BlockParameter`

Block parameters of the model that are modified.

**Variables — Variables**
array of `Simulink.Simulation.Variable`

Variables of the model that are modified.

**ModelParameters — Model parameters**
array of `Simulink.Simulation.ModelParameter`

Model parameters of the model that are modified.

**PreSimFcn — Function**
MATLAB function

MATLAB function to run before the start of the simulation.
**PostSimFcn — Function**

MATLAB function to run after the simulations.

**UserString — User string**

character array

Brief description of the simulation specified as a character array.

## Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>setModelParameter</td>
<td>Set model parameters to be used for a specific simulation through SimulationInput object.</td>
</tr>
<tr>
<td>setBlockParameter</td>
<td>Set block parameters to be used for a specific simulation through SimulationInput object.</td>
</tr>
<tr>
<td>setInitialState</td>
<td>Set initial state to be used for a specific simulation through SimulationInput object.</td>
</tr>
<tr>
<td>setExternalInput</td>
<td>Set external inputs for a simulation through SimulationInput object.</td>
</tr>
<tr>
<td>setVariable</td>
<td>Set variables for a simulation through SimulationInput object.</td>
</tr>
<tr>
<td>setPreSimFcn</td>
<td>Specify a MATLAB function to run before start of each simulation through SimulationInput object.</td>
</tr>
<tr>
<td>setPostSimFcn</td>
<td>Specify a MATLAB function to run after each simulation is complete through SimulationInput object.</td>
</tr>
<tr>
<td>applyToModel</td>
<td>Apply changes to the model specified through a SimulationInput object.</td>
</tr>
<tr>
<td>validate</td>
<td>Validate the contents of the SimulationInput object.</td>
</tr>
<tr>
<td>loadVariablesFromMATFile</td>
<td>Load variables from MAT-file into a Simulink.SimulationInput object.</td>
</tr>
</tbody>
</table>
Examples

Create a Simulink.SimulationInput Object

This example shows you how to create a SimulationInput object.

Open the model.

```matlab
openExample('simulink/OpenTheModelExample');
open_system('ex_sldemo_househeat');
load_system('ex_sldemo_househeat')
```

Create a single SimulationInput object for a model.

```matlab
model = 'ex_sldemo_househeat';
in = Simulink.SimulationInput(model);
```

Create an Array of Simulink.SimulationInput Objects

This example shows you how to create an array of SimulationInput objects.

Create an array of SimulationInput objects by using the `for` loop.

```matlab
model = 'vdp';
for i = 10:-1:1
    in(i) = Simulink.SimulationInput(model);[
end
```

Set Block Parameters using an Array of Simulink.SimulationInput Objects

This example modifies the block parameters of a model through the SimulationInput object.

Open the model.

```matlab
openExample('simulink/OpenTheModelExample');
open_system('ex_sldemo_househeat');
load_system('ex_sldemo_househeat')
```

Create a SimulationInput object for this model.
mdl = 'sldemo_househeat';
in = Simulink.SimulationInput(mdl);

Modify block parameter.
in = in.setBlockParameter('sldemo_househeat/Set Point','Value','300');

Simulate the model.
out = sim(in)

**Use Dataset as External Inputs with an Array of Simulink.SimulationInput Objects**

This example shows how use datasets to set external inputs with Simulink.SimulationInput objects.

Open the model

mdl = 'sldemo_mdlref_counter';
on_open_system(mdl);

Create a Dataset object for this model.

t = (0:0.01:10)';
ds{1} = timeseries(5*ones(size(t)),t);
ds{2} = timeseries(10*sin(t),t);
ds{3} = timeseries(-5*ones(size(t)),t);

Create a Simulink.SimulationInput object and set the external inputs

in = Simulink.SimulationInput('sldemo_mdlref_counter');
in = in.setExternalInput('ds.getElementById(1),ds.getElementById(2),ds.getElementById(3)');

Add other remaining variables in the expressions. Ensure that the dataset is also on the SimulationInput object.

Simulate the model.
in = in.setVariable('ds',ds);

Simulate the model
out = parsim(in)

See Also
Simulation Manager | Simulink.SimulationInput | applyToModel |
loadVariablesFromMATFile | parsim | setBlockParameter | setExternalInput |
setInitialState | setModelParameter | setPostSimFcn | setPreSimFcn |
setVariable | validate

Topics
“Run Multiple Simulations”
“Run Parallel Simulations Using parsim”

Introduced in R2017a
applyToModel

Apply changes to the model specified through a SimulationInput object, in

Syntax

in.applyToModel

Description

in.applyToModel applies the changes specified through the SimulationInput object to the model. You can use it to debug a model or to interactively analyze a simulation.

Examples

Apply Changes Made Through the Simulink.SimulationInput Object to the Model

This example shows how to modify a model through a SimulationInput object and save those modifications.

Open the model and create a SimulationInput object.

```
open_system('sldemo_househeat');
in = Simulink.SimulationInput('sldemo_househeat');
```

Modify block parameter, model parameters and a variable through SimulationInput object.

```
in = in.setBlockParameter('sldemo_househeat/Set Point','Value','75');
in = in.setVariable('cost',50,'Workspace','sldemo_househeat');
in = in.setModelParameter('StartTime','1','StopTime','5');
```

Apply the modifications made in the above step to the model.
in.applyToModel

See Also
Simulation Manager | Simulink.SimulationInput | loadVariablesFromMATFile | parsim | setBlockParameter | setExternalInput | setInitialState | setModelParameter | setPostSimFcn | setPreSimFcn | setVariable | validate

Topics
“Run Multiple Simulations”
“Run Parallel Simulations Using parsim”

Introduced in R2017a
removeVariable

Remove a variable from a Simulink.SimulationInput object or an array of Simulink.SimulationInput objects

Syntax

\[
in = \text{in}.\text{removeVariable}('\text{VariableName}')
\]
\[
in = \text{in}.\text{removeVariable}('\text{VariableName}', 'Workspace', 'ModelName')
\]

Description

\[
in = \text{in}.\text{removeVariable}('\text{VariableName}')\] removes all the variables with the name VariableName from the Simulink.SimulationInput object.

\[
in = \text{in}.\text{removeVariable}('\text{VariableName}', 'Workspace', 'ModelName')\] removes all the variables specified in the workspace by ModelName from the Simulink.SimulationInput object.

Examples

Modify and Remove a Variable for a Simulation

This example modifies the model parameters of through the SimulationInput object.

Open the model.

\[
\text{openExample}('\text{simulink/OpenTheModelExample}')
\]
\[
\text{open_system('ex_sldemo_househeat')}
\]
\[
\text{load_system('ex_sldemo_househeat')}
\]

Create a SimulationInput object for this model

\[
in = \text{Simulink.SimulationInput}('\text{mdl}')
\]

Set the variable value to 50.
in = in.setVariable('cost',50);

By default, this variable is placed in the global workspace scope.

Assume that you added the variable accidentally and want to remove it

in = in.removeVariable('cost');

### Input Arguments

'VariableName' — Name of variable

string

Name of the variable present in the Simulink.SimulationInput object

Example: 'Gain'

'Workspace' — Workspace for the variable

character vector

Set the scope of the defined variable by specifying the model name

Example: 'Workspace','sldemo_househeat'

### See Also

Simulation Manager | Simulink.SimulationInput | applyToModel | loadVariablesFromMATFile | parsim | setBlockParameter | setExternalInput | setInitialState | setModelParameter | setPostSimFcn | setPreSimFcn | validate

### Topics

“Run Multiple Simulations”
“Run Parallel Simulations Using parsim”

**Introduced in R2019a**
setExternalInput

Set external inputs for a simulation through SimulationInput object, in

Syntax

\[ \text{in} = \text{in.setExternalInput([t, u1,..uN])} \]
\[ \text{in} = \text{in.setExternalInput(ds)} \]
\[ \text{in} = \text{in.setExternalInput(ts)} \]

Description

\[ \text{in} = \text{in.setExternalInput([t, u1,..uN])} \] allows you to directly specify numerical arrays as inputs to a model if a model has root inports.

\[ \text{in} = \text{in.setExternalInput(ds)} \] allows you to directly specify dataset objects as external inputs to a model if a model has root inports.

\[ \text{in} = \text{in.setExternalInput(ts)} \] allows you to directly specify timeseries object as external input if a model has a single root inport.

Examples

Set Numerical Arrays as External Inputs Through a Simulink.SimulationInput Object

This example shows how to set numerical arrays as external inputs.

Open the model

\[ \text{open_system('sldemo_mdlref_counter');} \]

Create a SimulationInput object for this model.

\[ \text{in} = \text{Simulink.SimulationInput('sldemo_mdlref_counter');} \]
Prepare external inputs.

\[ t = (0:0.01:10)' \]
\[ u1 = 5*ones(size(t)); \]
\[ u2 = 10*sin(t); \]
\[ u3 = -5*ones(size(t)); \]

Set external inputs to the model.

\[ \text{in} = \text{in.setExternalInput([t, u1, u2, u3]);} \]

Simulate the model.

\[ \text{out} = \text{sim(in);} \]

**Input Arguments**

\[ [t, u1,..uN] \] — Numerical array

Numerical array

Numerical array to be used as an external input.

\[ \text{ds} \] — Dataset object

Simulink.SimulationData.Dataset object

Dataset object to be used as an external input

\[ \text{ts} \] — Time series

Time object handle

Time series to be used as an external input

**See Also**

Simulation Manager | Simulink.SimulationInput | applyToModel | loadVariablesFromMATFile | parsim | setBlockParameter | setInitialState | setModelParameter | setPostSimFcn | setPreSimFcn | setVariable | validate

**Topics**

“Run Multiple Simulations”
“Run Parallel Simulations Using parsim”

**Introduced in R2017a**
**setPostSimFcn**

Specify a MATLAB function to run after each simulation is complete through `SimulationInput` object, `in`

**Syntax**

```matlab
in = in.setPostSimFcn(@(y) myfunction(arg1, arg2 ...))
```

**Description**

`in = in.setPostSimFcn(@(y) myfunction(arg1, arg2 ...))` runs after each simulation is complete. The `Simulink.SimulationOutput` object is passed as the argument `y` to this function. `myfunction` is any MATLAB function and can be used to do the post processing on the output. To return post processed data, you must return it as values in a struct. These values are then packed into the `Simulink.SimulationOutput` output to replace the usual logged data or add new data to the `Simulink.SimulationOutput` object.

**Examples**

**Specify a MATLAB function for Postprocessing of the Output**

This example specifies a MATLAB Function through `SimulationInput` object to run after each simulation is complete.

Create a `PostSimFcn` to get the mean of the output.

```matlab
function newout = postsim(out);
newout.mean = mean(out.yout);
end
```

Create a `SimulationInput` object for a model.
in = Simulink.SimulationInput('vdp');
in = in.setPostSimFcn(@(x) postsim(x));
in = in.setModelParameter('SaveOutput','on');

Simulate the model.
out = sim(in)

View your result
out.mean

It is best practice to avoid using 'ErrorMessage' and 'SimulationMetadata' as field names in the function.

**Input Arguments**

`y` — Copy of Simulink.SimulationOutput object for postprocessing  
Simulink.SimulationOutput object

This is a Simulink.SimulationOutput object which is an input to myfunction.

`arg1, arg2 ...` — Function arguments  
character vectors object

Arguments specified to pass to the setPostSimFcn.

**See Also**

Simulation Manager | Simulink.SimulationInput | applyToModel | loadVariablesFromMATFile | parsim | setBlockParameter | setExternalInput | setInitialState | setModelParameter | setPreSimFcn | setVariable | validate

**Topics**

“Run Multiple Simulations”  
“Run Parallel Simulations Using parsim”

**Introduced in R2017a**
setPreSimFcn

Specify a MATLAB function to run before start of each simulation through SimulationInput object, in

Syntax

```matlab
in = in.setPreSimFcn(@(x) myfunction(arg1, arg2 ...))
```

Description

```matlab
in = in.setPreSimFcn(@(x) myfunction(arg1, arg2 ...))
```
runs before each simulation starts. The Simulink.SimulationInput object is passed as an argument x to this function. myfunction is any MATLAB function and can be used to modify the Simulink.SimulationInput object. If you use myfunction to modify the Simulink.SimulationInput object, you must return Simulink.SimulationInput object as the only output argument.

Examples

Specify a MATLAB Function to Run Before Each Simulation

This example shows how to specify a MATLAB function through SimulationInput object to run at before start of each simulation.

Create a PreSimFcn function.

```matlab
function presim(in)
    signalbuilder('sf_car/User Inputs', 'ActiveGroup', in.Variables.Value)
end
```

Open the model.

```matlab
model = 'sf_car';
open_system(model);
```
Create an array of `SimulationInput` objects for this model. Use `in.setPreSimFcn` to run `presim` before simulation.

```matlab
n = 4;
for idx = n:-1:1
    in(idx) = Simulink.SimulationInput(model);
    in(idx) = in(idx).setVariable('SigIndex', idx);
    in(idx) = in(idx).setPreSimFcn(@(x) presim(x));
end
```

Simulate the model.

```matlab
out = sim(in)
```

### Input Arguments

- **x** — A `Simulink.SimulationInput` object as input to the `myfunction`

  `Simulink.SimulationInput` object

  This is an input to `myfunction` in which you can modify the `Simulink.SimulationInput` object.

- **arg1, arg2 ...** — Function arguments

  character vectors object

  Arguments specified to pass to the `setPreSimFcn`.

### See Also

- `Simulation Manager`
- `Simulink.SimulationInput`
- `applyToModel`
- `loadVariablesFromMATFile`
- `parsim`
- `setBlockParameter`
- `setExternalInput`
- `setInitialState`
- `setModelParameter`
- `setPostSimFcn`
- `setVariable`
- `validate`

### Topics

- “Run Multiple Simulations”
- “Run Parallel Simulations Using `parsim`”

### Introduced in R2017a
**setBlockParameter**

Set block parameters to be used for a specific simulation through `SimulationInput` object, `in`.

**Syntax**

```matlab
in = in.setBlockParameter('BlockPath','ParameterName','Value',... 'BlockPathN''ParameterNameN','ValueN')
```

**Description**

`in = in.setBlockParameter('BlockPath','ParameterName','Value',... 'BlockPathN''ParameterNameN','ValueN')` sets the parameter on the block specified at `BlockPath` with the properties `ParameterName` and `Value`. You can use the `setBlockParameter` method to specify block parameters in model reference hierarchy. You can set multiple block parameters in a model using the same `SimulationInput` object. For more information on block parameter, see “Block-Specific Parameters” on page 6-132.

You can use `getBlockParameter('BlockPath','ParameterName')` method to get the value of block parameter and the `removeBlockParameter('BlockPath','ParameterName')` method to remove block parameter from the `Simulink.SimulationInput` object.

**Examples**

**Modify a Block Parameter for a Simulation**

This example modifies the block parameters of a model through the `SimulationInput` object.

Open the model.
mdl = 'sldemo_househeat';
open_system(mdl);

Create a SimulationInput object for this model.
in = Simulink.SimulationInput(mdl);

Modify block parameter.
in = in.setBlockParameter('sldemo_househeat/Set Point','Value','300');

Simulate the model.
out = sim(in)

Modify Multiple Block Parameters for a Simulation

This example modifies the block parameters of a model through the SimulationInput object.

Open the model
mdl = 'vdp';
open_system(mdl);

Create a SimulationInput object for this model.
in = Simulink.SimulationInput(mdl);

Modify block parameter.
in = in.setBlockParameter('vdp/Mu','Gain','40','vdp/Fcn','Position',[50 100 110 120]);

Simulate the model.
out = sim(in)

Input Arguments

'BlockPath' — Path of the block
character vector
BlockPath is the path of the block for which the parameter is changed
Example: 'sldemo_househeat/Set Point'

'ParameterName' — Block parameter name
character vector

Specify optional comma-separated pairs of ParameterName, Value arguments. ParameterName is the parameter name and Value is the corresponding value. ParameterName must appear inside single quotes (''). Block parameter values are typically specified as character vectors. You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN. ParameterNameN,ValueN pairs follow the same syntax as set_param.
Example: 'Value', '350'

See Also
Simulation Manager | Simulink.SimulationInput | applyToModel | loadVariablesFromMATFile | parsim | setExternalInput | setInitialState | setModelParameter | setPostSimFcn | setPreSimFcn | setVariable | validate

Topics
“Run Multiple Simulations”
“Run Parallel Simulations Using parsim”

Introduced in R2017a
setInitialState

Set initial state to be used for a specific simulation through SimulationInput object, in

Syntax

\[ \text{in} = \text{in.setInitialState}(\text{xInitial}) \]

Description

\[ \text{in} = \text{in.setInitialState}(\text{xInitial}) \] sets the initial state of a model to \( x_{\text{Initial}} \), a Simulink.op.ModelOperatingPoint object.

Input Arguments

\( x_{\text{Initial}} \) — Simulink.op.ModelOperatingPoint object

Simulink.op.ModelOperatingPoint

You can change the Initial State of a model by assigning it to a Simulink.op.ModelOperatingPoint object.

See Also

Simulation Manager | Simulink.SimulationInput | applyToModel |
loadVariablesFromMATFile | parsim | setBlockParameter | setExternalInput |
setModelParameter | setPostSimFcn | setPreSimFcn | setVariable | validate

Topics

“Run Multiple Simulations”
“Run Parallel Simulations Using parsim”

Introduced in R2017a
**setModelParameter**

Set model parameters to be used for a specific simulation through `SimulationInput` object, `in`

**Syntax**

```matlab
in = in.setModelParameter('ParameterName', 'Value', ...
    'ParemeterNameN', 'ValueN')
```

**Description**

```matlab
in = in.setModelParameter('ParameterName', 'Value', ...
    'ParemeterNameN', 'ValueN')
```

sets a model parameter `Name` with a `Value`. You can add multiple model parameters to the model using the same `SimulationInput` object. For more information on model parameters, see “Model Parameters” on page 6-2.

You can use `getModelProperty('ParameterName')` method to get the value of model parameter and the `removeModelProperty('ParameterName')` method to remove model parameter from the `Simulink.SimulationInput` object.

**Examples**

**Modify a Model Parameter for a Simulation**

This example modifies the model parameters of through the `SimulationInput` object.

Open the model.

```matlab
mdl = 'sldemo_househeat';
open_system(mdl);
```

Create a `SimulationInput` object for this model.

```matlab
in = Simulink.SimulationInput(mdl);
```
Specify a timeout of 5 seconds and modify model parameters, StartTime and StopTime

```matlab
in = in.setModelParameter('Timeout', 5);
in = in.setModelParameter('StartTime', '1', 'StopTime', '4');
```

Simulate the model.

```matlab
out = sim(in)
```

### Input Arguments

- `'ParameterName', 'Value', ... 'ParameterNameN', 'ValueN'` — Block parameter name

  Specify optional comma-separated pairs of `ParameterName,Value` arguments. `ParameterName` is the parameter name and `Value` is the corresponding value. `ParameterName` must appear inside single quotes ('). Model parameter values are typically specified as character vectors. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`. `ParameterNameN,ValueN` pairs follow the same syntax as `set_param`.

  Example: `'StartTime','1'`

### See Also

- `Simulation Manager`
- `Simulink.SimulationInput`
- `applyToModel`
- `loadVariablesFromMATFile`
- `parsim`
- `setBlockParameter`
- `setExternalInput`
- `setInitialState`
- `setPostSimFcn`
- `setPreSimFcn`
- `setVariable`
- `validate`

### Topics

- “Run Multiple Simulations”
- “Run Parallel Simulations Using parsim”

### Introduced in R2017a
**setVariable**

Set variables for a simulation through `SimulationInput` object, in

**Syntax**

```plaintext
in = in.setVariable('Name', 'Value')
in = in.setVariable('Name', 'Value','Workspace', 'ModelName')
```

**Description**

```plaintext
in = in.setVariable('Name', 'Value') assigns a Value to variable Name. You can add multiple variables to the model using the same `SimulationInput` object.

in = in.setVariable('Name', 'Value','Workspace', 'ModelName') assigns the Value to variable Name. Variables that are defined through the `SimulationInput` object are placed in the global workspace scope by default. The term global workspace is specific to the Simulink `SimulationInput` object and its methods. Variables in the global workspace scope take precedence if a variable with the same name exists in the base workspace or the data dictionary. The variables in the model workspace take precedence over the global workspace scope. To change the value of a model workspace variable, set the scope by specifying the model name when you add the variable to the `SimulationInput` object.

While using the `setVariable` method, you can specify only one variable at a time.

You can use `getVariable('VariableName')` method to get the value of variable and the `removeVariable('VariableName')` method to remove variable from the `SimulationInput` object.

For information on using nonscalar variables, structure variables and parameter objects, see “Sweep Nonscalars, Structures, and Parameter Objects”.

**Examples**
**Modify a Variable for a Simulation**

This example modifies the model parameters of through the `SimulationInput` object.

Open the model.

```matlab
mdl = 'sldemo_househeat';
open_system(mdl);
```

Create a `SimulationInput` object for this model

```matlab
in = Simulink.SimulationInput(mdl);
```

Set the variable value to 50.

```matlab
in = in.setVariable('cost',50);
```

By default, this variable is placed in the global workspace scope.

Simulate the model.

```matlab
out = sim(in)
```

**Modify a Variable for a Simulation in the Model Workspace**

This example modifies the model parameters of `sldemo_househeat` through the `SimulationInput` object.

Set path and open the model.

```matlab
mdl = 'sldemo_househeat';
open_system(mdl);
```

Create a `SimulationInput` object for this model

```matlab
in = Simulink.SimulationInput(mdl);
```

Set the variable to 50 and set the scope to model workspace.

```matlab
in = in.setVariable('cost',50,'Workspace','sldemo_househeat');
```

Simulate the model.
out = sim(in)

**Input Arguments**

`'Name', 'Value' — Block parameter name  
character vector

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the literal value of the variable. `Name` must appear inside single quotes (`' '`).

Example: `'StartTime','1'

`'Workspace' — Workspace for the variable 
character vector

Set the scope of the defined variable by specifying the model name

Example: `'Workspace','sldemo_househeat'

**See Also**

Simulation Manager | `Simulink.SimulationInput` | `applyToModel` | `loadVariablesFromMATFile` | `parsim` | `setBlockParameter` | `setExternalInput` | `setInitialState` | `setModelParameter` | `setPostSimFcn` | `setPreSimFcn` | `validate`

**Topics**

“Run Multiple Simulations”
“Run Parallel Simulations Using parsim”

**Introduced in R2017a**
**validate**

Validate the contents of the `SimulationInput` object, `in`

**Syntax**

`in.validate`

**Description**

`in.validate` validates the changes made to the model through the `SimulationInput` object. This method validates all the changes made to the model through the `SimulationInput` object.

**Examples**

**Validate Changes Made Through the SimulationInput Object**

This example modifies and validates the variable of the model through the `SimulationInput` object.

Open the model.

```matlab
mdl = 'sldemo_househeat';
on_system(mdl);
```

Create a `SimulationInput` object for this model

```matlab
in = Simulink.SimulationInput(mdl);
```

Modify a model parameter

```matlab
in = in.setModelParameter('InvalidParamName','5');
```

Validate this change
in.validate

**See Also**
Simulation Manager | Simulink.SimulationInput | applyToModel | loadVariablesFromMATFile | parsim | setBlockParameter | setExternalInput | setInitialState | setModelParameter | setPostSimFcn | setPreSimFcn | setVariable

**Topics**
“Run Multiple Simulations”
“Run Parallel Simulations Using parsim”

**Introduced in R2017a**
loadVariablesFromMATFile

Load variables from MAT-file into a Simulink.SimulationInput object, in

Syntax

\[ \text{in} = \text{in}.\text{loadVariablesFromMATFile}(\text{'filename'}) \]

Description

\[ \text{in} = \text{in}.\text{loadVariablesFromMATFile}(\text{'filename'}) \] loads variables from the specified MAT-file into the Variables property of Simulink.SimulationInput.

Examples

Load Variables From MAT-File

This example shows how to load variables from MAT-File into Simulink.SimulationInput object using the loadVariablesFromMATFile method.

Create a Simulink.SimulationInput Object

Create a Simulink.SimulationInput object for the example model.

\[
\text{mdl} = \text{'ex_loadVar_sldemo_househeat'}; \\
\text{in} = \text{Simulink.SimulationInput(mdl)}; \\
\text{in}
\]

\[
in = \\
\text{SimulationInput with properties:} \\
\text{ModelName: 'ex_loadVar_sldemo_househeat'} \\
\text{InitialState: [0x0 Simulink.op.ModelOperatingPoint]} \\
\text{ExternalInput: []}
\]
Load Variables From MAT-File

Use the `loadVariablesFromMATFile` method of the `Simulink.SimulationInput` object.

```matlab
in = in.loadVariablesFromMATFile('sldemo_househeat_data.mat');
in
in =

SimulationInput with properties:
  ModelName: 'ex_loadVar_sldemo_househeat'
  InitialState: [0x0 Simulink.op.ModelOperatingPoint]
  ExternalInput: []
  ModelParameters: [0x0 Simulink.Simulation.ModelParameter]
  BlockParameters: [0x0 Simulink.Simulation.BlockParameter]
  Variables: [1x25 Simulink.Simulation.Variable]
  PreSimFcn: []
  PostSimFcn: []
  UserString: ''
```

Simulate the Model

The model uses the loaded variables for simulation

```matlab
out = sim(in)
```

```matlab
out =

Simulink.SimulationOutput:
  sldemo_househeat_output: [1x1 Simulink.SimulationData.Dataset]
    SimulationMetadata: [1x1 Simulink.SimulationMetadata]
```
Input Arguments

filename — Name of the MAT file
character vector | string scalar

Name of MAT-file, specified as a character vector or string scalar.
Example: 'myFile'

See Also
Simulation Manager | Simulink.SimulationInput | applyToModel | parsim |
setBlockParameter | setExternalInput | setInitialState |
setModelParameter | setPostSimFcn | setPreSimFcn | setVariable

Topics
“Run Multiple Simulations”
“Run Parallel Simulations Using parsim”

Introduced in R2017a
Simulink.Simulation.Futures class

**Package:** Simulink

Create Future object for simulation

**Description**

Create a Simulink.Simulation.Future object when you execute parsim with the 'RunInBackground' argument set to 'on'. The parsim command runs multiple simulations in parallel using the inputs specified with an array of Simulink.SimulationInput objects. You can use this object to monitor the status of ongoing simulations, fetch outputs of completed simulations, or cancel ongoing simulations.

The parsim command uses the Parallel Computing Toolbox license to run the simulations in parallel. parsim runs the simulations in serial if a parallel pool cannot be created or if Parallel Computing Toolbox is not used.

**Construction**

future = parsim(in,'RunInBackground','on') creates a Simulink.Simulation.Future object, future, while running multiple simulations in parallel the using the inputs specified in the Simulink.SimulationInput object, in.

**Input Arguments**

in — Simulink.SimulationInput object array
   object (default) | array

A Simulink.SimulationInput object or an array of Simulink.SimulationInput objects is used to run multiple simulations. Specify parameters and values of a model to run multiple simulations without making it dirty.

Example: in = Simulink.SimulationInput('vdp'), in(1:10) = Simulink.SimulationInput('vdp')
'RunInBackground' — parsim argument to enable RunInbackground
'off' (default) | 'on'

Set to 'on', to run simulations asynchronously, keeping the MATLAB command prompt available.

Properties

Diary — Log of outputs from the simulation
text file

Text log of outputs from the simulation.

This property is read-only.

ID — Numeric identifier of the future object
scalar integer

ID of the future object, specified as a scalar integer.

This property is read-only.

Read — Whether the outputs have been read
1 | 0

Whether a call to fetchNext or fetchOutputs has read the outputs in the Simulink.Simulation.Future object array, specified as 1 if true and 0 if false.

This property is read-only.

State — Current state of future object array
'pending' | 'queued' | 'running' | 'finished' | 'failed' | 'unavailable'

Current state of future object array, specified as 'pending', 'queued', 'running', 'finished', 'failed', or 'unavailable'.

This property is read-only.
Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>cancel</td>
<td>Cancel a pending, queued, or running Simulink.Simulation.Future object</td>
</tr>
<tr>
<td>fetchNext</td>
<td>Fetch next available unread output from Simulink.Simulation.Future object array</td>
</tr>
<tr>
<td>fetchOutputs</td>
<td>Retrieve Simulink.SimulationOutput from Simulink.Simulation.Future</td>
</tr>
<tr>
<td>wait</td>
<td>Wait for Simulink.Simulation.Future objects to complete simulation</td>
</tr>
</tbody>
</table>

Examples

Create a Simulink.Simulation.Future Object

This example shows how to create a Simulink.Simulation.Future object array and use it to retrieve outputs and see the status of simulations.

This example runs several simulations of the vdp model, varying the value of the gain Mu.

Open the model and define a vector of Mu values.

```matlab
open_system('vdp');
Mu_Values = [0.5:0.25:5];
MuVal_length = length(Mu_Values);
```

Using Mu_Values, initialize an array of Simulink.SimulationInput objects. To preallocate the array, a loop index is made to start from the largest value.

```matlab
for i = MuVal_length:-1:1
    in(i) = Simulink.SimulationInput('vdp');
    in(i) = in(i).setBlockParameter('vdp/Mu',...
        'Gain',num2str(Mu_Values(i)));
end
```

Simulate the model using parsim. Set it to 'RunInBackground', to be able to use the command prompt, while simulations are running.
Future = parsim(in, 'RunInBackground', 'on');

Use the fetchNext method on Future simulations.

for i = 1:MuVal_length
    [completedIdx, simOut] = fetchNext(Future)
end

See Also

Functions
batch | batchsim | cancel | fetchNext | fetchOutputs | parsim | wait

Classes
Simulink.SimulationInput

Introduced in R2018a
cancel

Cancel a pending, queued, or running Simulink.Simulation.Future object

Syntax

cancel(Future)

Description

cancel(Future) stops the objects of the Simulink.Simulation.Future array, Future, that are currently in 'pending', 'queued', or 'running' state. For elements of the Futures in the 'finished' state, no action is taken.

Examples

Cancel Simulations of Future Objects

This example shows how to use the cancel method on an array of future objects to stop the simulations.

This example runs several simulations of the vdp model, varying the value of the gain Mu.

Open the model and define a vector of Mu values.

open_system('vdp');
Mu_Values = [0.5:0.25:1000];
MuVal_length = length(Mu_Values)

Using Mu_Values, initialize an array of Simulink.SimulationInput objects. To preallocate the array, a loop index is made to start from the largest value.

for i = MuVal_length:-1:1
    in(i) = Simulink.SimulationInput('vdp');
    in(i) = in(i).setBlockParameter('vdp/Mu',...
end

Simulate the model using parsim. Set to 'RunInBackground' to enable the use the command prompt, while simulations are running.

Future = parsim(in,'RunInBackground','on');

Now, assume that you want to run simulations with different values of Mu and cancel the ongoing simulations.

cancel(Future)

**Input Arguments**

**Future** — Simulation.Simulink.Future object array

Array of Simulation.Simulink.Future objects. To create Future, run parsim with 'RunInBackground' option set to 'on'.

Example: `Future = parsim(in,'RunInBackground','on')`

**See Also**

**Functions**

`batch | batchsim | fetchNext | fetchOutputs | parsim | wait`

**Classes**

`Simulink.Simulation.Future | Simulink.SimulationInput`

**Introduced in R2018a**
fetchNext

Fetch next available unread output from Simulink.Simulation.Future object array

Syntax

[ idx, simOut ] = fetchNext( Future )
[ idx, simOut ] = fetchNext( Future, Timeout )

Description


[ idx, simOut ] = fetchNext( Future, Timeout ) waits for a maximum of Timeout seconds for a result to become available. If the timeout expires before any result is available, simOut is returned as an empty array.

An error is reported if there are no elements in Future with property Read as false. You can check for are any unread futures using anyUnread = ~all([F.Read]).

fetchNext displays an error if any element of Future with a 'finished' state encounters an error during execution. The Read property of that element becomes true allowing any subsequent call to fetchNext to proceed.

Examples

Create a Simulink.Simulation.Future Object

This example shows how to create a Simulink.Simulation.Future object array and use it to retrieve outputs and see the status of simulations.

This example runs several simulations of the vdp model, varying the value of the gain Mu.
Open the model and define a vector of Mu values.

```matlab
open_system('vdp');
Mu_Values = [0.5:0.25:5];
MuVal_length = length(Mu_Values);
```

Using Mu_Values, initialize an array of Simulink.SimulationInput objects. To preallocate the array, a loop index is made to start from the largest value.

```matlab
for i = MuVal_length:-1:1
    in(i) = Simulink.SimulationInput('vdp');
    in(i) = in(i).setBlockParameter('vdp/Mu', ...
        'Gain',num2str(Mu_Values(i)));
end
```

Simulate the model using `parsim`. Set it to 'RunInBackground' to be able to use the command prompt, while simulations are running.

```matlab
Future = parsim(in,'RunInBackground','on');
```

Use the `fetchNext` method on Future.

```matlab
for i = 1:MuVal_length
    [completedIdx,simOut] = fetchNext(Future)
end
```

**Input Arguments**

- **Future** — `Simulation.Simulink.Future` object array
  
  Array of `Simulation.Simulink.Future` objects. To create `Future`, run `parsim` with 'RunInBackground' option set to 'on'.
  
  Example: `Future = parsim(in,'RunInBackground','on')`

- **Timeout** — Number of seconds specified for `fetchNext` to time out scalar
  
  Specify a Timeout for `fetchNext` to retrieve the results from the `Simulation.Simulink.Future` array, `Future`.
  
  Example: `[idx, simOut] = fetchNext(Future, 45)`
Output Arguments

**idx — Index of the simulation**
integer

When `fetchNext` method is used on an array of `Simulink.Simulation.Future` objects, it returns the index of the simulation whose output is being retrieved.

**simOut — Simulation object containing logged simulation results**
object | array

Array of `Simulink.SimulationOutput` objects that contain all of the logged simulation results. The size of the array is equal to the size of the array of `Simulink.SimulationInput` objects.

All simulation outputs (logged time, states, and signals) are returned in a single `Simulink.SimulationOutput` object. You define the model time, states, and output that are logged using the **Data Import/Export** pane of the Model Configuration Parameters dialog box. You can log signals using blocks such as the To Workspace and Scope blocks. The **Signal & Scope Manager** tool can directly log signals.

See Also

**Functions**
batch | batchsim | cancel | fetchOutputs | parsim | wait

**Classes**
`Simulink.Simulation.Future` | `Simulink.SimulationInput`

**Introduced in R2017b**
fetchOutputs

Retrieve Simulink.SimulationOutput from Simulink.Simulation.Future objects

Syntax

simOut = fetchOutputs(Future)

Description

simOut = fetchOutputs(Future) fetches the output from an array of Simulink.Simulation.Future objects, Future, after each element of Future is in a 'finished' state. fetchOutputs returns an array of Simulink.SimulationOutput objects.

Examples

Create a Future and Retrieve Outputs Using fetchOutputs

This example shows how to use the fetchOutputs method on an array of future objects to retrieve a Simulink.SimulationOutput array.

This example runs several simulations of the vdp model, varying the value of the gain Mu.

Open the model and define a vector of Mu values.

```matlab
open_system('vdp');
Mu_Values = [0.5,0.75,1,1.25];
MuVal_length = length(Mu_Values);
```

Using Mu_Values, initialize an array of Simulink.SimulationInput objects. To preallocate the array, a loop index is made to start from the largest value.

```matlab
for i = MuVal_length:-1:1
    in(i) = Simulink.SimulationInput('vdp');
```
in(i) = in(i).setBlockParameter('vdp/Mu',...
   'Gain',num2str(Mu_Values(i)));
end

Simulate the model using parsim. Set to 'RunInBackground' to enable the use of command prompt, while simulations are running.

Future = parsim(in,'RunInBackground','on');

Use the fetchOutputs method on Future

simOut = fetchOutputs(Future)

simOut =

1x4 Simulink.SimulationOutput array

Input Arguments

Future — Simulation.Simulink.Future object array

Array of Simulation.Simulink.Future objects. To create, Future, run parsim with 'RunInBackground' option set to 'on'.

Example: Future = parsim(in,'RunInBackground','on')

Output Arguments

simOut — Simulation object containing logged simulation results object

Array of Simulink.SimulationOutput objects that contain all of the logged simulation results. The size of the array is equal to the size of the array of Simulink.SimulationInput objects.

All simulation outputs (logged time, states, and signals) are returned in a single Simulink.SimulationOutput object. You define the model time, states, and output that are logged using the Data Import/Export pane of the Model Configuration Parameters dialog box. You can log signals using blocks such as the To Workspace and Scope blocks. The Signal & Scope Manager tool can directly log signals.
See Also

Functions
batch | batchsim | cancel | fetchNext | parsim | wait

Classes
Simulink.Simulation.Future | Simulink.SimulationInput

Introduced in R2018a
**wait**

Wait for `Simulink.Simulation.Future` objects to complete simulation.

**Syntax**

```matlab
Ok = wait(Future)
```

**Description**

`Ok = wait(Future)` blocks the command prompt until each element of the `Simulink.Simulation.Future` array, `Future` is in a 'finished' state.

**Examples**

**Wait for the Future Array to Complete Simulations**

This example shows how to use the `wait` method on an array of future objects.

This example runs several simulations of the `vdp` model, varying the value of the gain `Mu`.

Open the model and define a vector of `Mu` values.

```matlab
open_system('vdp');
Mu_Values = [0.5:0.25:5];
MuVal_length = length(Mu_Values);
```

Using `Mu_Values`, initialize an array of `Simulink.SimulationInput` objects. To preallocate the array, a loop index is made to start from the largest value.

```matlab
for i = MuVal_length:-1:1
    in(i) = Simulink.SimulationInput('vdp');
    in(i) = in(i).setBlockParameter('vdp/Mu', ...
        'Gain',num2str(Mu_Values(i)));
end
```
Simulate the model using `parsim`. Set to 'RunInBackground' to enable the use of command prompt while the simulations are running and to create an array of `Simulink.Simulation.Future` objects.

```matlab
Future = parsim(in,'RunInBackground','on');
```

Use the `wait` method on `Future` to block the execution.

```matlab
Ok = wait(Future)
```

### Input Arguments

**Future** — `Simulation.Simulink.Future` object array

Array of `Simulation.Simulink.Future` objects. To create `Future`, run `parsim` with 'RunInBackground' option set to 'on'.

Example: `Future = parsim(in,'RunInBackground','on')`

### Output Arguments

**Ok** — Whether the wait is completed successfully

1 | 0

`Ok` is `true` if the wait completes successfully, `false` if any of the `Future` objects failed execution or were canceled. Specified as `1` if true, `0` if false.

### See Also

**Functions**

`batch | batchsim | cancel | fetchNext | fetchOutputs | parsim`

**Classes**

`Simulink.Simulation.Future | Simulink.SimulationInput`

**Introduced in R2017b**
Simulink.SimulationMetadata class

**Package:** Simulink

Access metadata of simulation runs

**Description**

The `SimulationMetadata` class contains information about a simulation run including:

- Model information
- Timing information
- Execution and diagnostic information
- Custom character vector to tag the simulation
- Custom data to describe the simulation

`SimulationMetadata` packages this information with the `SimulationOutput` object. To use `SimulationMetadata`, use one of these approaches:

- In `Configuration Parameters` > `Data Import/Export`, under `Save options`, select `Single simulation output`.
- Use `set_param` to set `ReturnWorkspaceOutputs` to on.

```matlab
set_param(model_name,'ReturnWorkspaceOutputs','on');
```

To retrieve the `SimulationMetadata` object, use the `getSimulationMetadata` method on a `SimulationOutput` object.

**Properties**

**ModelInfo** — Information about the model and simulation operating environment

*structure*

The `ModelInfo` structure has these fields.
<table>
<thead>
<tr>
<th><strong>Field Name</strong></th>
<th><strong>Type</strong></th>
<th><strong>Description</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>ModelName</td>
<td>char</td>
<td>Name of the model</td>
</tr>
<tr>
<td>ModelVersion</td>
<td>char</td>
<td>Version of the model</td>
</tr>
<tr>
<td>ModelFilePath</td>
<td>char</td>
<td>Absolute location of the .mdl/.slx file</td>
</tr>
<tr>
<td>UserID</td>
<td>char</td>
<td>System user ID of the machine used for the simulation</td>
</tr>
<tr>
<td>MachineName</td>
<td>char</td>
<td>Hostname of the machine used for the simulation</td>
</tr>
<tr>
<td>Platform</td>
<td>char</td>
<td>Operating system of the machine used for the simulation</td>
</tr>
<tr>
<td>ModelStructuralChecksum</td>
<td>4-by-1 uint32</td>
<td>Structural checksum of the model calculated after an update diagram</td>
</tr>
<tr>
<td>SimulationMode</td>
<td>char</td>
<td>Simulation mode</td>
</tr>
<tr>
<td>StartTime</td>
<td>double</td>
<td>Simulation start time</td>
</tr>
<tr>
<td>StopTime</td>
<td>double</td>
<td>Time at which the simulation was terminated</td>
</tr>
<tr>
<td>SolverInfo</td>
<td>structure</td>
<td>Solver information:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Fixed-step solvers – Solver type, name, and fixed step size</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Variable solvers – Solver type, name, and max step size (initial setting)</td>
</tr>
<tr>
<td>SimulinkVersion</td>
<td>structure</td>
<td>Version of Simulink</td>
</tr>
<tr>
<td>LoggingInfo</td>
<td>structure</td>
<td>Metadata about logging to persistent storage:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• LoggingToFile field — Indicates whether logging to persistent storage is enabled ('on' or 'off')</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• LoggingFileName field — Specifies the resolved file name for the persistent storage MAT-file (if LoggingToFile is 'on').</td>
</tr>
</tbody>
</table>
ExecutionInfo — Structure to store information about a simulation run
structure

Structure to store information about a simulation run, including the reason a simulation stopped and any diagnostics reported during the simulation. The structure has these fields.
<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>StopEvent</td>
<td>Nontranslated character vector</td>
<td>Reason the simulation stopped, represented by one of the following: \n  - ReachedStopTime - Simulation stopped upon reaching stop time and no errors were reported during execution. StopEvent has value ReachedStopTime, even if errors are reported in the stop callbacks, which are executed after the simulation ends. \n  - ModelStop - Simulation stopped by a block or by solver before reaching stop time. \n  - StopCommand - Simulation stopped manually by clicking the Stop button or using the set_param command. \n  - DiagnosticError - Simulation stopped because an error was reported during simulation. \n  - KeyboardControlC - Simulation stopped using keystroke Ctrl+C. \n  - PauseCommand - Simulation paused manually by clicking the Pause button or using the set_param command. \n  - ConditionalPause - Simulation paused using a conditional breakpoint. \n  - PauseTime - Simulation paused at or after specified pause time. \n  - StepForward - Simulation paused after clicking step forward. \n  - StepBackward - Simulation paused after clicking step backward.</td>
</tr>
<tr>
<td>Field Name</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>---------------</td>
<td>-------------------------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• <strong>TimeOut</strong> – Simulation stopped because execution time exceeded timeout specified by TimeOut.</td>
</tr>
<tr>
<td>StopEventSource</td>
<td>Simulink.SimulationData BlockPath</td>
<td>Source of stop event, if it is a valid Simulink object.</td>
</tr>
<tr>
<td>StopEventDescription</td>
<td>Translated character vector</td>
<td>Superset of information stored in StopEvent and StopEventSource.</td>
</tr>
<tr>
<td>ErrorDiagnostic</td>
<td>struct</td>
<td>Error reported during simulation, represented by the following fields:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• <strong>Diagnostic</strong> – MSLDiagnostic object that includes object paths, ID, message, cause, and stack.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• <strong>SimulationPhase</strong> – Represented by one of these: Initialization, Execution, or Termination.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• <strong>SimulationTime</strong> – Simulation time represented as a double, if reported during Execution; else, represented as [].</td>
</tr>
</tbody>
</table>

By passing the name-value pair 'CaptureErrors', 'on' to the sim command, errors generated during simulation are reported in ExecutionInfo.ErrorDiagnostic. The sim command does not capture generated errors.
### WarningDiagnostics

Array of all warnings reported during the simulation. Each array item is represented by the following fields:

- **Diagnostic** - MSLDiagnostic object that includes object paths, ID, message, cause, and stack.
- **SimulationPhase** - Represented as: Initialization, Execution, or Termination.
- **SimulationTime** - Simulation time represented as a double, if reported during Execution; else, represented as [].

### TimingInfo — Structure to store profiling information about the simulation

Structure to store profiling information about the simulation, including the time stamps for the start and end of the simulation. The structure has these fields.

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WallClockTimestampStart</td>
<td>character vector</td>
<td>Wall clock time when the simulation started, in YYYY-MM-DD HH:MI:SS format with microsecond resolution</td>
</tr>
<tr>
<td>WallClockTimestampStop</td>
<td>character vector</td>
<td>Wall clock time when the simulation stopped, in YYYY-MM-DD HH:MI:SS format with microsecond resolution</td>
</tr>
<tr>
<td>InitializationElapsedWallTime</td>
<td>double</td>
<td>Time spent before execution, in seconds</td>
</tr>
<tr>
<td>ExecutionElapsedWallTime</td>
<td>double</td>
<td>Time spent during execution, in seconds</td>
</tr>
<tr>
<td>TerminationElapsedWallTime</td>
<td>double</td>
<td>Time spent after execution, in seconds</td>
</tr>
<tr>
<td>Field Name</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------</td>
<td>------------</td>
<td>------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>TotalElapsedWallTime</td>
<td>double</td>
<td>Total time spent in initialization, execution, and termination, in seconds</td>
</tr>
</tbody>
</table>

The `ExecutionElapsedWallTime` includes the time that Simulink spent to roll back or step back in a simulation. The `ExecutionElapsedWallTime` does not include the time spent between steps. For example, if you use Stepper to step through a simulation, the `ExecutionElapsedWallTime` time does not include the time when the simulation is in a paused state. For more information about using Stepper, see “How Simulation Stepper Helps With Model Analysis”.

**UserString** — Custom character vector to describe the simulation

character vector

Use `Simulink.SimulationOutput.setUserString` to directly store a character vector in the `SimulationMetadata` object that is contained in the `SimulationOutput` object.

**UserData** — Custom data to store in `SimulationMetadata` object that is contained in the `SimulationOutput` object

character vector

Use `Simulink.SimulationOutput.setUserData` to store custom data in the `SimulationMetadata` object that is contained in the `SimulationOutput` object.

**Copy Semantics**

Value. To learn how value classes affect copy operations, see Copying Objects (MATLAB).

**Examples**

**Get a SimulationMetadata Object for vdp Simulation**

Simulate the `vdp` model. Retrieve metadata from a `SimulationMetadata` object of the simulation.

Simulate the `vdp` model. Save the results of the `Simulink.SimulationOutput` object in `simout`.  

5-822
open_system('vdp');
simout = sim(bdroot,'ReturnWorkspaceOutputs','on');
Retrieve metadata information about this simulation using `mData`. This is the `SimulationMetadata` object that `simout` contains.

```matlab
mData = simout.getSimulationMetadata()
```

```matlab
mData = 
    SimulationMetadata with properties:
        ModelInfo: [1x1 struct]
        TimingInfo: [1x1 struct]
        ExecutionInfo: [1x1 struct]
        UserString: ''
        UserData: []
```
Store custom data or string in simout.

```matlab
simout = simout.setUserData(struct('param1','value1','param2','value2','param3','value3'));
simout = simout.setUserString('Store first simulation results');
```

Retrieve the custom data you stored from mData.

```matlab
mData = simout.getSimulationMetadata()
disp(mData.UserData)
```

```matlab
mData =

SimulationMetadata with properties:

    ModelInfo: [1x1 struct]
    TimingInfo: [1x1 struct]
    ExecutionInfo: [1x1 struct]
    UserString: 'Store first simulation results'
    UserData: [1x1 struct]

    param1: 'value1'
    param2: 'value2'
    param3: 'value3'
```

Retrieve the custom string you stored from mData.

```matlab
mData = simout.getSimulationMetadata()
disp(mData.UserString)
```

```matlab
mData =

SimulationMetadata with properties:

    ModelInfo: [1x1 struct]
    TimingInfo: [1x1 struct]
    ExecutionInfo: [1x1 struct]
    UserString: 'Store first simulation results'
    UserData: [1x1 struct]
```
Store first simulation results

See Also
Simulink.SimulationOutput.getSimulationMetadata|
Simulink.SimulationOutput.setUserData|
Simulink.SimulationOutput.setUserString

Introduced in R2015a
Simulink.SimulationOutput class

**Package:** Simulink

Access object values of simulation results

**Description**

The `SimulationOutput` class contains all simulation outputs, including workspace variables.

You can use dot notation to access the data for simulation outputs. For example, to return data for the xout variable for a simOut, `SimulationOutput` object, use a `simOut.tout` command.

Alternatively, you can use `Simulink.SimulationOutput.who` and either `Simulink.SimulationOutput.get` or `Simulink.SimulationOutput.find` methods to access the output variable names and their respective values.

The `SimulationOutput` object is writeable. You can write to `logsout` and `yout` in `SimulationOutput` object.

You can use the `plot` function to plot the `SimulationOutput` output.

```
plot(out)
```

**Properties**

**SimulationMetadata — Metadata for simulation runs**

`Simulink.SimulationMetadata` object

Metadata for simulation runs, returned as a `Simulink.SimulationMetadata` object. Fields other than the `UserData` and `UserString` fields are read only.

**ErrorMessage — Simulation logging error messages**

`char` vector

Simulation logging error message, returned as a char vector. (read only)
## Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>find</td>
<td>Access and display values of simulation results</td>
</tr>
<tr>
<td>get</td>
<td>Access and display values of simulation results</td>
</tr>
<tr>
<td>getSimulationMetadata</td>
<td>Return SimulationMetadata object for simulation</td>
</tr>
<tr>
<td>setUserData</td>
<td>Store custom data in SimulationMetadata object that SimulationOutput object contains</td>
</tr>
<tr>
<td>setUserString</td>
<td>Store custom character vector in SimulationMetadata object that SimulationOutput object contains</td>
</tr>
<tr>
<td>who</td>
<td>Access and display output variable names of simulation</td>
</tr>
<tr>
<td>plot</td>
<td>Plot simulation output data in the Simulation Data Inspector</td>
</tr>
</tbody>
</table>

## Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects (MATLAB) in the MATLAB Programming Fundamentals documentation.

## Examples

### View Simulation Output and Metadata

Simulate a model and place the results of the `Simulink.SimulationOutput` object in `simOut` and view the simulation metadata.

Simulate the `vdp` model.

```matlab
simOut = sim('vdp','SimulationMode','normal','AbsTol','1e-5',... 'SaveState','on','StateSaveName','xoutNew',... 'SaveOutput','on','OutputSaveName','youtNew')
```

`Simulink.SimulationOutput`:

- `xoutNew: [65x2 double]`
- `youtNew: [65x2 double]`
Get the values of the variable `youtNew`.

```matlab
simOut.youtNew
```

Simulink returns and displays the values.

Get the timing information for the simulation.

```matlab
myMetadata = simOut.SimulationMetadata
```

```matlab
myMetadata =
```

```matlab
SimulationMetadata with properties:
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```{markdown}
### See Also

`Simulink.SimulationData.Dataset`, `loadIntoMemory`  

### Topics

“Migrate Scripts That Use Legacy ModelDataLogs API”  
“Export Simulation Data”
```
Simulink.SubsysDataLogs

Container for subsystem signal data logs

Description

**Note** Before R2016a, the `Simulink.SubsysDataLogs` class was used in conjunction with the `ModelDataLogs` logging data format. Starting in R2016a, you cannot log data in the `ModelDataLogs` format. Signal logging uses the `Dataset` format.

However, you can use data that was logged in a previous release using `ModelDataLogs` format.

In releases before R2016a, Simulink created instances of this class to contain logs for signals in a subsystem models were logged in `ModelDataLogs` format. Objects of this class have a variable number of properties. The first property, named `Name`, is the name of the subsystem whose log data this object contains. The remaining properties are signal log or signal log container objects containing the data logged for the subsystem specified by this object's `Name` property.

For example, suppose you have this logged data from a model run in a release earlier than R2016a:

<table>
<thead>
<tr>
<th>Simulink.SubsysDataLogs (Gain):</th>
<th>elements</th>
<th>Simulink Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>1</td>
<td>Timeseries</td>
</tr>
<tr>
<td>m</td>
<td>2</td>
<td>TsArray</td>
</tr>
</tbody>
</table>

You can use either fully qualified log names or the `unpack` command to access the signal logs contained by a `SubsysDataLogs` object. For example, to access the amplitudes logged for signal `a` in the preceding example, you could enter the following at the MATLAB command line:

```matlab
data = logsout.Gain.a.Data;
```

or
>> logsout.unpack('all');
data = a.Data;

See Also


Introduced before R2006a
Simulink.SuppressedDiagnostic class

**Package:** Simulink

Suppress diagnostic messages from a specified block

**Description**

A Simulink.SuppressedDiagnostic object contains information related to diagnostic warnings or errors that are suppressed from being thrown during simulation.

**Construction**

The Diagnostic Viewer in Simulink includes an option to suppress certain diagnostics. This feature enables you to suppress warnings or errors for specific objects in your model. Click the **Suppress** button next to the warning in the Diagnostic Viewer to suppress the warning from the specified source. This action creates a Simulink.SuppressedDiagnostic object. You can access this object at the MATLAB command line using the Simulink.getSuppressedDiagnostics function. You can add a comment for the suppressed diagnostic. You can restore the diagnostic by clicking **Restore**.

```plaintext
DiagnosticObject = Simulink.SuppressedDiagnostic(source, message_id)
```

creates a suppressed diagnostic object. The object suppresses all instances of diagnostics represented by `message_id` thrown by the blocks specified by `source`. 
**Input Arguments**

**source** — System, block, or model object throwing diagnostic  
model | subsystem | block path | block handle

The source of the diagnostic, specified as a model, subsystem, block path, block handle, cell array of block paths, or cell array of block handles.

To get the block path, use the `gcb` function.

To get the block handle, use the `getSimulinkBlockHandle` function.

Data Types: char | cell | string

**message_id** — message identifier of diagnostic  
message identifier

The message identifier of the diagnostic, specified as a character vector or string. You can find the message identifier of diagnostics thrown during simulation by accessing the `ExecutionInfo` property of the `Simulink.SimulationMetadata` object associated with a simulation. You can also use the `lastwarn` function.

Data Types: char | string

**Properties**

**Comments** — Comments associated with the suppression object  
character vector

Comments associated with the suppression object, specified as a character vector. This property is optional.

Data Types: char

**ID** — Message identifier of the diagnostic that was suppressed  
character vector

The message identifier of the diagnostic that was suppressed, specified as a character vector.

Data Types: char
**LastModified — Date and time the suppression object was last modified**
character vector

Date and time the suppression object was last modified, specified as a character vector. This property is read-only.

Data Types: char

**LastModifiedBy — Name of the user who was last to add or edit the suppression object**
character vector

Name of the user who last added or edited the suppression object, specified as a character vector. This property is optional.

Data Types: char

**Source — block path of the source of the diagnostic**
character vector

The block path of the model object that has a suppressed diagnostic, specified as a character vector.

Data Types: char

**Methods**

- `restore` Remove specified diagnostic suppressions
- `suppress` Suppress diagnostic specified by `Simulink.SuppressedDiagnostic` object

**Copy Semantics**

Handle. To learn how handle classes affect copy operations, see Copying Objects (MATLAB).

**Examples**
Create a Simulink.SuppressedDiagnostic Object

Using the model from “Suppress Diagnostic Messages Programmatically”, create and then restore a diagnostic suppression.

Create a Simulink.SuppressedDiagnostic object, suppression to suppress the parameter precision loss warning thrown by the Constant block, one.

```matlab
suppression = Simulink.SuppressedDiagnostic('Suppressor_CLI_Demo/one', ... 'SimulinkFixedPoint:util:fxpParameterPrecisionLoss');
```

The parameter precision loss warning is no longer thrown in future simulations of this model.

Add accountability information to the object by editing the LastModifiedBy and Comments properties of the object.

```matlab
suppression.LastModifiedBy = 'John Doe';
suppression.Comments = 'Reviewed: Joe Schmoe'
```

```matlab
suppression =

  SuppressedDiagnostic with properties:

  Source: 'Suppressor_CLI_Demo/one'
  Id: 'SimulinkFixedPoint:util:fxpParameterPrecisionLoss'
  LastModifiedBy: 'John Doe'
  Comments: 'Reviewed: Joe Schmoe'
  LastModified: '2016-Jun-01 17:25:21'
```

You can restore the diagnostic using the restore method.

```matlab
restore(suppression);
```

See Also

Simulink.SuppressedDiagnostic | Simulink.SuppressedDiagnostic.restore | Simulink.getSuppressedDiagnostics | Simulink.restoreDiagnostic | Simulink.suppressDiagnostic

Topics

“Suppress Diagnostic Messages Programmatically”
Class Attributes (MATLAB)
Property Attributes (MATLAB)

Introduced in R2016b
restore

Class: Simulink.SuppressedDiagnostic
Package: Simulink

Remove specified diagnostic suppressions

Syntax

restore(SuppressedDiagnostic)

Description

restore(SuppressedDiagnostic) removes the specified suppressed diagnostic object.

Input Arguments

SuppressedDiagnostic — Suppressed diagnostic object to restore
Simulink.SuppressedDiagnostic object

Examples

Restore a Suppressed Diagnostic

Using the model from “Suppress Diagnostic Messages Programmatically”, create and then restore a diagnostic suppression.

Create a Simulink.SuppressedDiagnostic object, suppression to suppress the parameter precision loss warning from the Constant block, one.
suppression = Simulink.SuppressedDiagnostic('Suppressor_CLI_Demo/one', ...
'SimulinkFixedPoint:util:fxpParameterPrecisionLoss');

You can restore the diagnostic using the restore method.

restore(suppression);

**Restore All Suppressed Diagnostics**

Using the model from “Suppress Diagnostic Messages Programmatically”, restore all diagnostic suppressions associated with a model.

Use the Simulink.suppressDiagnostic function to suppress the parameter precision loss and parameter underflow warnings from the Constant block, one.

diags = {'SimulinkFixedPoint:util:fxpParameterPrecisionLoss', 'SimulinkFixedPoint:util:fxpParameterUnderflow'};
Simulink.suppressDiagnostic('Suppressor_CLI_Demo/one',diags);

Use the Simulink.getSuppressedDiagnostics function to get all suppressions associated with the model, returned as an array of Simulink.SuppressedDiagnostic objects.

suppressed_diagnostics = Simulink.getSuppressedDiagnostics('Suppressor_CLI_Demo')

suppressed_diagnostics =

    1x2 SuppressedDiagnostic array with properties:
    Source
    Id
    LastModifiedBy
    Comments
    LastModified

Restore all diagnostics using the restore method and iterating through the suppressed_diagnostics array.
for iter = 1:numel(suppressed_diagnostics)
    restore(suppressed_diagnostics(iter));
end

See Also
Simulink.SuppressDiagnostic | Simulink.getSuppressedDiagnostics | Simulink.restoreDiagnostic | Simulink.suppressDiagnostic

Topics
“Suppress Diagnostic Messages Programmatically”

Introduced in R2016b
**suppress**

**Class:** Simulink.SuppressedDiagnostic  
**Package:** Simulink

Suppress diagnostic specified by Simulink.SuppressedDiagnostic object

**Syntax**

```plaintext
suppress(SuppressedDiagnostic)
```

**Description**

`suppress(SuppressedDiagnostic)` suppresses the specified suppressed diagnostic object.

**Input Arguments**

- **SuppressedDiagnostic** — Suppressed diagnostic object to suppress  
  Simulink.SuppressedDiagnostic object

**See Also**

- Simulink.SuppressedDiagnostic  
- Simulink.getSuppressedDiagnostics  
- Simulink.restoreDiagnostic  
- Simulink.suppressDiagnostic

**Topics**

- “Suppress Diagnostic Messages Programmatically”

**Introduced in R2018a**
Simulink.TimeInfo

Provide information about time data in Simulink.Timeseries object

Description

Simulink software creates instances of these objects to describe the time data that it includes in Simulink.Timeseries objects.

Note The Simulink.Timeseries class is supported for backwards compatibility. The ModelDataLogs format created Simulink.Timeseries objects for signal logging data. Starting in R2016a, you cannot log data in the ModelDataLogs format. Signal logging uses the Dataset format. In R2016a or later, when you open a model from an earlier release that had used ModelDataLogs format, the model simulated in use Dataset format.

You can convert signal logging data from ModelDataLogs to Dataset format. Converting to Dataset format makes it easier to post-process with other logged data (for example, logged states), which can also use Dataset format. For more information, see “Convert Logged Data to Dataset Format”.

If you have legacy code that uses the ModelDataLogs API, you can encounter situations that require updates to your code or model. See “Migrate Scripts That Use Legacy ModelDataLogs API”.

Properties

<table>
<thead>
<tr>
<th>Name</th>
<th>Access</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Units</td>
<td>RW</td>
<td>The units, e.g., 'seconds', in which the time series data are expressed in the associated Simulink.Timeseries object.</td>
</tr>
<tr>
<td>Name</td>
<td>Access</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>--------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Start</td>
<td>RW</td>
<td>If the associated signal is not in a conditionally executed subsystem, this field contains the simulation time of the first signal value recorded in the associated Simulink.Timeseries object. If the signal is in a conditionally executed subsystem, this field contains an array of times when the system became active.</td>
</tr>
<tr>
<td>end</td>
<td>RW</td>
<td>If the associated signal is not in a conditionally executed subsystem, this field contains the simulation time of the last signal value recorded in the associated Simulink.Timeseries object. If the signal is in a conditionally executed subsystem, this field contains an array of times when the system became inactive.</td>
</tr>
<tr>
<td>Increment</td>
<td>RW</td>
<td>The interval between simulation times at which signal data is logged in the associated Simulink.Timeseries object. If the signal is aperiodic (continuous signal with variable-step solver), this property has a value of NaN. A signal is periodic if it has a discrete sample time (not continuous or constant) or is continuous with a fixed-step solver.</td>
</tr>
<tr>
<td>Length</td>
<td>W</td>
<td>The number of signal samples recorded in the associated Simulink.Timeseries object, i.e., the length of the arrays referenced by the object's Time and Data properties.</td>
</tr>
</tbody>
</table>

**See Also**

Simulink.Timeseries, Simulink.SimulationData.Dataset

**Introduced before R2006a**
Simulink.Timeseries

Store data for any signal except mux or bus signal

Description

**Note** The Simulink.Timeseries class is supported for backwards compatibility. The ModelDataLogs format created Simulink.Timeseries objects for signal logging data. Starting in R2016a, you cannot log data in the ModelDataLogs format. Signal logging uses the Dataset format. In R2016a or later, when you open a model from an earlier release that had used ModelDataLogs format, the model simulated in use Dataset format.

You can convert signal logging data from ModelDataLogs to Dataset format. Converting to Dataset format makes it easier to post-process with other logged data (for example, logged states), which can also use Dataset format. For more information, see “Convert Logged Data to Dataset Format”.

If you have legacy code that uses the ModelDataLogs API, you can encounter situations that require updates to your code or model. See “Migrate Scripts That Use Legacy ModelDataLogs API”.

Simulink software creates instances of this class to store signal data that it logs for any signal except a mux or bus signal, which are stored in a Simulink.TsArray. See “Export Signal Data Using Signal Logging” for more information.

Properties

<table>
<thead>
<tr>
<th>Name</th>
<th>Access</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>RW</td>
<td>Name of this signal log.</td>
</tr>
<tr>
<td>BlockPath</td>
<td>RW</td>
<td>Path of the block that output the signal logged in this signal log.</td>
</tr>
<tr>
<td>Name</td>
<td>Access</td>
<td>Description</td>
</tr>
<tr>
<td>-------------</td>
<td>--------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>PortIndex</td>
<td>RW</td>
<td>Index of the output port that emitted the signal logged in this signal log.</td>
</tr>
<tr>
<td>SignalName</td>
<td>RW</td>
<td>Name of the signal logged in this signal log.</td>
</tr>
<tr>
<td>ParentName</td>
<td>RW</td>
<td>Name of the parent of the signal recorded in this log, if the signal is an element of a mux or a virtual bus; otherwise, the same as SignalName.</td>
</tr>
<tr>
<td>TimeInfo</td>
<td>RW</td>
<td>An object of Simulink.TimeInfo class that describes the time data in this log.</td>
</tr>
<tr>
<td>Time</td>
<td>RW</td>
<td>An array containing the simulation times at which signal data was logged.</td>
</tr>
<tr>
<td>Data</td>
<td>RW</td>
<td>An array containing the signal data.</td>
</tr>
</tbody>
</table>

**See Also**


**Introduced before R2006a**
Simulink.TsArray

Store data for mux or bus signal

Description

**Note** Before R2016a, the Simulink.TsArray class was used in conjunction with the ModelDataLogs logging data format. Starting in R2016a, you cannot log data in the ModelDataLogs format. Signal logging uses the Dataset format.

However, you can use data that was logged in a previous release using ModelDataLogs format.

In releases earlier than R2016a, Simulink software created instances of this class to contain the data that it logs for a mux or bus signal. Other types of signals were stored in a Simulink.Timeseries.

Objects of the Simulink.TsArray class have a variable number of properties. The first property, called Name, specifies the log name of the logged signal. The remaining properties reference logs for the elements of the logged signal: Simulink.Timeseries objects for elementary signals and Simulink.TSArray objects for mux or bus signals. The name of each property is the log name of the corresponding signal.

For example, suppose you have this logged data from a model run in a release earlier than R2016a that was configured to log in ModelDataLogs format.

```plaintext
logouts.b2
Simulink.TsArray (untitled/Bus Creator1):
  Name     elements  Simulink Class
           x1          1      Timeseries
           b1          2      TsArray
```

The Simulink.ModelDataLogs object, named logouts, contains a Simulink.TsArray object, named b2, that contains the logs for the elements of b2 (that is, the elementary signal x1 and the bus signal b1). Entering the fully qualified name of
the Simulink.TsArray object, (logout.b2) at the MATLAB command line reveals the structure of the signal log for this model.

You can use either fully qualified log names or the unpack command to access the signal logs contained by a Simulink.TsArray object. For example, to access the amplitudes logged for signal x1 in the preceding example, you can enter the following at the MATLAB command line:

data = logout.b2.x1.Data;

or

logout.unpack('all');
data = x1.Data;

See Also


Introduced before R2006a
Simulink.Variant class

Package: Simulink

Specify conditions that control variant selection

Description

An object of the Simulink.Variant class represents a conditional expression called a variant control. The object allows you to specify a Boolean expression that activates a specific variant choice when it evaluates to true.

A variant control comprises one or more variant control variables, specified using MATLAB variables or Simulink.Parameter objects.

You specify variant controls for each variant choice represented in a Variant Subsystem or Model Variant block. For a given Variant Subsystem or Model Variant block, only one variant control can evaluate to true at a time. When a variant control evaluates to true, Simulink activates the variant choice that corresponds to that variant control.

Construction

variantControl = Simulink.Variant(conditionExpression) creates a variant control.

Properties

conditionExpression — Variant condition expression

'(default)' | character vector

Variant condition expression, specified as a character vector containing one or more of these operands and operators.

Operands
Variable names that resolve to MATLAB variables or Simulink.Parameter objects with integer or enumerated data type and scalar literal values
• Variable names that resolve to Simulink.Variant objects
• Scalar literal values that represent integer or enumerated values

Operators
• Parentheses for grouping
• Arithmetic, relational, logical, or bit-wise operators

The variant condition expression evaluates to a Boolean value. This property has read and write access.

Example: 'Fuel==2 || Emission==1 && Ratio==2'

Examples

Create Variant Controls Using MATLAB Variables

Use MATLAB variables when you want to simulate the model but are not considering code generation.

Create MATLAB variables with scalar literal values.

\[
\begin{align*}
Fuel &= 3; \\
Emission &= 1; \\
Ratio &= 3;
\end{align*}
\]

Develop conditional expressions using the variables.

\[
\begin{align*}
Variant1 &= \text{Simulink.Variant}('Fuel==1 && Emission==2'); \\
Variant2 &= \text{Simulink.Variant}('(Fuel==2 || Emission==1) && Ratio==2'); \\
Variant3 &= \text{Simulink.Variant}('Fuel==3 || Ratio==4');
\end{align*}
\]

Create Variant Controls Using Simulink.Parameter Objects

If you want to generate preprocessor conditionals for code generation, use Simulink.Parameter objects.

Create variant Simulink.Parameter objects with scalar literal values.
Fuel = Simulink.Parameter(3);
Emission = Simulink.Parameter(1);
Ratio = Simulink.Parameter(3);

Specify the custom storage class for these objects as ImportedDefine so that the values are specified by an external header file.

Other valid values for the custom storage class are Define and CompilerFlag.

Fuel.CoderInfo.StorageClass = 'Custom';
Fuel.CoderInfo.CustomStorageClass = 'ImportedDefine';

Emission.CoderInfo.StorageClass = 'Custom';
Emission.CoderInfo.CustomStorageClass = 'ImportedDefine';

Ratio.CoderInfo.StorageClass = 'Custom';
Ratio.CoderInfo.CustomStorageClass = 'ImportedDefine';

Develop conditional expressions using the variables and create variant controls.

Variant1 = Simulink.Variant('Fuel==1 && Emission==2');
Variant2 = Simulink.Variant('(Fuel==2 || Emission==1) && Ratio==2');
Variant3 = Simulink.Variant('Fuel==3 || Ratio==4');

See Also
“Operators and Operands in Variant Condition Expressions”

Topics
“Define, Configure, and Activate Variants”
“Convert Variant Control Variables into Simulink.Parameter Objects”
“Approaches for Specifying Variant Controls”
Simulink.VariantConfigurationData class

Package: Simulink

Class representing a variant configurations data object

Description

The variant configuration data object, stores a collection of variant configurations, constraints, and the name of the default active configuration. The Simulink.VariantConfigurationData class has properties that enable you to add, modify, or remove variant configurations, constraints, and control variables. Use an instance of Simulink.VariantConfigurationData class to do the following:

- Define and edit variant configurations.
- Add control variables to variant configurations.
- Add copy of variant configuration.
- Delete existing variant configurations, constraints, and sub model configurations.
- Set a specific configuration as default active.
- Validate model using default or a specific variant configuration.
- Query or create variant configurations data object for a given model.

Properties

VariantConfigurations

Set of variant configurations. The names of the configurations must be unique and valid MATLAB variable names.

Constraints

Set of constraints that must always be satisfied by the model for all variant configurations. The name of the constraints must be unique and valid MATLAB variable names.
**DefaultConfigurationName**

Name of the variant configuration to be used by default for validation.
## Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>addConfiguration</td>
<td>Add a new variant configuration to the variant configuration data object</td>
</tr>
<tr>
<td>addConstraint</td>
<td>Add a constraint to the variant configuration data object</td>
</tr>
<tr>
<td>addControlVariables</td>
<td>Add control variables to an existing variant configuration</td>
</tr>
<tr>
<td>addCopyOfConfiguration</td>
<td>Add a copy of an existing variant configuration to the variant configuration data object</td>
</tr>
<tr>
<td>addSubModelConfigurations</td>
<td>Add to a variant configuration the names of the configurations to be used for submodels</td>
</tr>
<tr>
<td>existsFor</td>
<td>Check if variant configuration data object exists for a model</td>
</tr>
<tr>
<td>getConfiguration</td>
<td>Returns the variant configuration with a given name from a variant configuration data object</td>
</tr>
<tr>
<td>getDefaultValue</td>
<td>Returns default variant configuration, if any, for a variant configuration data object</td>
</tr>
<tr>
<td>getFor</td>
<td>Get existing variant configuration data object for a model</td>
</tr>
<tr>
<td>getOrCreateFor</td>
<td>Get existing or create a new variant configuration data object for a model</td>
</tr>
<tr>
<td>removeConfiguration</td>
<td>Remove a variant configuration with a given name from the variant configuration data object</td>
</tr>
<tr>
<td>removeConstraint</td>
<td>Remove a constraint from the variant configuration data object</td>
</tr>
<tr>
<td>removeControlVariable</td>
<td>Remove a control variable from a variant configuration</td>
</tr>
<tr>
<td>removeSubModelConfiguration</td>
<td>Remove from a variant configuration, the configuration to be used for a sub model.</td>
</tr>
<tr>
<td>setDefaultConfigurationName</td>
<td>Set name of the default variant configuration for a variant configuration data object</td>
</tr>
<tr>
<td>validateModel</td>
<td>Validate all variant blocks in the model and submodels in the hierarchy during simulation</td>
</tr>
<tr>
<td>VariantConfigurationData</td>
<td>Object constructor with optional arguments for variant configurations, constraints, and default configuration name</td>
</tr>
</tbody>
</table>
Examples

load_system(model);
% Create variant config and associate it with model
variantConfig = Simulink.VariantConfigurationData;
set_param(model, 'VariantConfigurationObject', 'variantConfig');

See Also

Topics

“Variant Manager Overview”
addConfiguration

**Class:** Simulink.VariantConfigurationData  
**Package:** Simulink

Add a new variant configuration to the variant configuration data object

**Syntax**

```matlab
vcdataObj.addConfiguration(name)
vcregObj.addConfiguration(name,description)
vcregObj.addConfiguration(name,description,controlVars)
vcregObj.addConfiguration(name,description,controlVars,
subModelConfigurations)
```

**Description**

`vcdataObj.addConfiguration(name)` adds a new variant configuration with a given name to the variant configuration data object.

`vcdataObj.addConfiguration(name,description)` adds a new variant configuration with a given name and optional description to the variant configuration data object.

`vcdataObj.addConfiguration(name,description,controlVars)` adds a new variant configuration with a given name, optional description, and control variables to the variant configuration data object.

`vcdataObj.addConfiguration(name,description,controlVars,
subModelConfigurations)` adds a new variant configuration with a given name, optional description, control variables, and submodel configurations to the variant configuration data object.
Input Arguments

name

Name of variant configuration being added.

description

Description text for the variant configuration being added.

controlVars

Control variables for the variant configuration being added. This argument must be a vector of structures with required fields: Name and Value. The values assigned to the Name field must be unique and valid MATLAB variable names. The Value field can contain either character vectors or Simulink.Parameter objects. The values of control variables are checked during validation of the variant configuration.

subModelConfigurations

Vector of structures containing fields: ModelName, ConfigurationName. The names of submodels must be unique and valid MATLAB variable names and configuration names must be valid MATLAB variables.

Examples

% Define the variant configuration data object
vcdataObj = Simulink.VariantConfigurationData;

% Add a variant configuration LinInterExp
vcdataObj.addConfiguration('LinInterExp')

See Also
Simulink.VariantConfigurationData |
Simulink.VariantConfigurationData.addControlVariables |
Simulink.VariantConfigurationData.addSubModelConfigurations
addConstraint

Class: Simulink.VariantConfigurationData
Package: Simulink

Add a constraint to the variant configuration data object

Syntax

vcdataObj.addConstraint(nameOfConstraint)
vcdataObj.addConstraint(nameOfConstraint,condition)
vcdataObj.addConstraint(nameOfConstraint,condition,description)

Description

vcdataObj.addConstraint(nameOfConstraint) adds a new constraint with a given name to vcdataObj.

vcdataObj.addConstraint(nameOfConstraint,condition) adds a new constraint with a given name and condition expression to vcdataObj.

vcdataObj.addConstraint(nameOfConstraint,condition,description) adds a new constraint with a given name, condition expression, and description to vcdataObj.

Input Arguments

nameOfConstraint
Name of constraint being added. Must be unique and valid MATLAB variable name.

condition
Boolean expression that must evaluate to true. When the expression evaluates to true, it means the constraint is satisfied.
description

Text that describes the constraint.

Examples

% Define the variant configuration data object
vcdataObj = Simulink.VariantConfigurationData;

% Add a constraint named LinNotExtern
vcdataObj.addConstraint('LinNotExtern','((Ctrl~=1)... || (PlantLocation ~=1))','Description of the constraint')

See Also
Simulink.VariantConfigurationData|
Simulink.VariantConfigurationData.addConfiguration |
Simulink.VariantConfigurationData.removeConfiguration |
Simulink.VariantConfigurationData.removeConstraint
addControlVariables

Class: Simulink.VariantConfigurationData
Package: Simulink

Add control variables to an existing variant configuration

Syntax

vcdataObj.addControlVariables(nameOfConfiguration,controlVars)

Description

vcdataObj.addControlVariables(nameOfConfiguration,controlVars), adds control variables to a variant configuration.

Input Arguments

nameOfConfiguration

Specifies the name of an existing configuration.

controlVars

Control variables being added. This argument must be a vector of structures with required fields: Name and Value. The values assigned to the Name field must be unique and valid MATLAB variable names. The Value field can contain either character vectors or Simulink.Parameter objects. The values of control variables are checked during validation of the variant configuration.

Examples

% Define the variant configuration data object
vcdataObj = Simulink.VariantConfigurationData;
% Add a variant configuration named LinInterExp
vcdataObj.addConfiguration('LinInterExp', ...
'Linear Internal Experimental Plant Controller');

% Add control variables SmartSensor1Mod and PlanLocation
vcdataObj.addControlVariables('LinInterExp', ...
    cell2struct({{'SmartSensor1Mod', '2';...
            'PlantLocation', '1'}},...
    {'Name', 'Value'}, 2))

See Also
Simulink.VariantConfigurationData |
Simulink.VariantConfigurationData.addSubModelConfigurations |
Simulink.VariantConfigurationData.removeControlVariable |
Simulink.VariantConfigurationData.removeSubModelConfiguration
addCopyOfConfiguration

**Class:** Simulink.VariantConfigurationData  
**Package:** Simulink

Add a copy of an existing variant configuration to the variant configuration data object

**Syntax**

```java
vcdataObj.addCopyOfConfiguration(nameOfExistingConfiguration)
vcdataObj.addCopyOfConfiguration(nameOfExistingConfiguration, nameOfTobeAddedConfiguration)
```

**Description**

vcdataObj.addCopyOfConfiguration(nameOfExistingConfiguration), adds a new configuration with a default name (default name is based on existing configuration name being copied) as a copy of the existing configuration to the variant configuration data object.

vcdataObj.addCopyOfConfiguration(nameOfExistingConfiguration, nameOfTobeAddedConfiguration), adds a new configuration with a specified name, as a copy of the existing configuration, to the variant configuration data object.

**Input Arguments**

**nameOfExistingConfiguration**

Name of existing configuration.

**Default:**

**nameOfTobeAddedConfiguration**

Name of new configuration to be added as a copy of the configuration.
Examples

% Define the variant configuration data object
vcdataObj = Simulink.VariantConfigurationData;

% Add the variant configuration LinInterExp
vcdataObj.addConfiguration('LinInterExp', ...
   'Linear Internal Experimental Plant Controller');

% Add a copy of variant configuration LinInterExp
% and name the copy as LinExtExp
vcdataObj.addCopyOfConfiguration('LinInterExp','LinExtExp')

See Also
Simulink.VariantConfigurationData |
Simulink.VariantConfigurationData.addConfiguration |
Simulink.VariantConfigurationData.removeConfiguration |
Simulink.VariantConfigurationData.setDefaultConfiguration
addSubModelConfigurations

Class: Simulink.VariantConfigurationData
Package: Simulink

Add to a variant configuration the names of the configurations to be used for submodels

Syntax

vcdataObj.addSubModelConfigurations(nameOfConfiguration, subModelConfigurations)

Description

vcdataObj.addSubModelConfigurations(nameOfConfiguration, subModelConfigurations), specifies names of the configurations to be used for submodels.

Input Arguments

nameOfConfiguration

Name for the configuration of submodels that are model references.

subModelConfigurations

Vector of structures containing fields: ModelName, ConfigurationName. The names of submodels must be unique and valid MATLAB variable names and configuration names must be valid MATLAB variables.

Examples

1    Add the path to the model file.

   addpath(fullfile(docroot,'toolbox','simulink','examples'));
2 Load the model.

    load_system('slexVariantManagementExample');

3 Add a new submodel configuration to LinInterExpNoNoise.

    vcdataObj.addSubModelConfigurations('LinInterExpNoNoise',...  
        [struct('ModelName', 'slexVariantManagementExternalPlantMdlRef',...  
            'ConfigurationName', 'LowFid')])

**See Also**

Simulink.VariantConfigurationData |  
Simulink.VariantConfigurationData.addControlVariables |  
Simulink.VariantConfigurationData.removeControlVariable |  
Simulink.VariantConfigurationData.removeSubModelConfiguration
existsFor

**Class:** Simulink.VariantConfigurationData  
**Package:** Simulink

Check if variant configuration data object exists for a model

**Syntax**

Simulink.VariantConfigurationData.existsFor(modelNameOrHandle)

**Description**

Simulink.VariantConfigurationData.existsFor(modelNameOrHandle) returns true if the variant configuration data object exists for the model.

**Input Arguments**

modelNameOrHandle

Name or handle to the model.

**Examples**

```matlab
% Add the path to the model file
addpath(fullfile(docroot,'toolbox','simulink','examples'));

% Load the model
load_system('slexVariantManagementExample');

% Checks whether a variant configuration data object exists for model
[exists] = Simulink.VariantConfigurationData.existsFor...
('slexVariantManagementExample')
```
See Also
Simulink.VariantConfigurationData
Simulink.VariantConfigurationData.existsFor
getConfiguration

**Class:** Simulink.VariantConfigurationData  
**Package:** Simulink

Returns the variant configuration with a given name from a variant configuration data object.

**Syntax**

vcdataObj.getConfiguration(nameOfConfiguration)

**Description**

vcdataObj.getConfiguration(nameOfConfiguration) returns a specific variant configuration that is associated with the variant configuration data object.

**Input Arguments**

nameOfConfiguration

Name of the variant configuration to be returned.

**Examples**

% Define the variant configuration data object
vcdataObj = Simulink.VariantConfigurationData;

% Add the variant configuration LinInterExp
vcdataObj.addConfiguration('LinInterExp',...
   'Linear Internal Experimental Plant Controller');

% Add a control variable SmartSensor1Mod
vcdataObj.addControlVariables('LinInterExp',...
   [struct('Name','SmartSensor1Mod','Value','2')]);
% Obtain information on the variant configuration.
% LinInterExp from the variant configuration data object
vc = vcdataObj.getConfiguration('LinInterExp')

See Also
Simulink.VariantConfigurationData |
Simulink.VariantConfigurationData.addConfiguration |
Simulink.VariantConfigurationData.getDefaultConfiguration |
Simulink.VariantConfigurationData.removeConfiguration
getDefaul tConfiguration

**Class:** Simulink.VariantConfigurationData  
**Package:** Simulink

Returns default variant configuration, if any, for a variant configuration data object

**Syntax**

vcdataObj.getDefaultConfiguration

**Description**

vcdataObj.getDefaultConfiguration returns the default variant configuration. If no default variant configuration is defined, then [] is returned.

**Examples**

```matlab
% Define the variant configuration data object
vcdataObj = Simulink.VariantConfigurationData;

% Add the variant configuration named LinInterExp
vcdataObj.addConfiguration('LinInterExp',...
    'Linear Internal Experimental Plant Controller');

% Add the variant configuration LinInterStd
vcdataObj.addConfiguration('LinInterStd',...
    'Linear Internal Standard Plant Controller');

% Set LinExtExp as the default variant configuration
vcdataObj.setDefaultConfigurationName('LinExtExp');

% Obtain the default variant configuration
defvc = vcdataObj.getDefaultConfiguration
```
See Also
Simulink.VariantConfigurationData |
Simulink.VariantConfigurationData.getConfiguration |
Simulink.VariantConfigurationData.setDefaultConfigurationName
getFor

Class: Simulink.VariantConfigurationData
Package: Simulink

Get existing variant configuration data object for a model

Syntax

Simulink.VariantConfigurationData.getFor(modelNameOrHandle)

Description

Simulink.VariantConfigurationData.getFor(modelNameOrHandle), returns the variant configuration object for the model. If no default variant configuration is defined, then [] is returned.

Input Arguments

modelNameOrHandle

Model name or handle.

Examples

% Add the path to the model file
addpath(fullfile(docroot,'toolbox','simulink','examples'));

% Load the model
load_system('slexVariantManagementExample');

% Obtain variant configuration data object for the model
% slexVariantManagementExample
vcdataObj = Simulink.VariantConfigurationData.getFor...
('slexVariantManagementExample')
See Also

Simulink.VariantConfigurationData |
Simulink.VariantConfigurationData.existsFor |
Simulink.VariantConfigurationData.getOrCreateFor
getOrCreateFor

**Class:** Simulink.VariantConfigurationData  
**Package:** Simulink  

Get existing or create a new variant configuration data object for a model

**Syntax**

Simulink.VariantConfigurationData.getOrCreateFor(modelNameOrHandle)

**Description**

Simulink.VariantConfigurationData.getOrCreateFor(modelNameOrHandle), returns the object if the variant configuration data objects exists otherwise, creates an empty object.

**Input Arguments**

**modelNameOrHandle**  
Model name or handle to the model.

**Examples**

```matlab
% Add the path to the model file  
addpath(fullfile(docroot,'toolbox','simulink','examples'));

% Load the model  
load_system('slexVariantManagementExample');

% Obtain existing or create an empty variant configuration  
% data object for the slexVariantManagementExample model  
vcdatadobj = Simulink.VariantConfigurationData.getOrCreateFor...  
   ('slexVariantManagementExample')
```
See Also
Simulink.VariantConfigurationData|
Simulink.VariantConfigurationData.existsFor|
Simulink.VariantConfigurationData.getFor
removeConfiguration

Class: Simulink.VariantConfigurationData
Package: Simulink

Remove a variant configuration with a given name from the variant configuration data object

Syntax

vcdataObj.removeConfiguration(nameOfConfiguration)

Description

vcdataObj.removeConfiguration(nameOfConfiguration) removes the configuration from the variant configuration data object.

Input Arguments

nameOfConfiguration

Name of the configuration to be removed.

Examples

% Define the variant configuration data object
vcdataObj = Simulink.VariantConfigurationData;

% Add the LinInterExp variant configuration
% to the variant configuration data object
vcdataObj.addConfiguration('LinInterExp',...
   'Linear Internal Experimental Plant Controller');

% Remove the LinInterExp configuration
% from the variant configuration data object
vcdataObj.removeConfiguration('LinInterExp')

See Also
Simulink.VariantConfigurationData |
Simulink.VariantConfigurationData.addConfiguration |
Simulink.VariantConfigurationData.getConfiguration
removeConstraint

Class: Simulink.VariantConfigurationData
Package: Simulink

Remove a constraint from the variant configuration data object

Syntax

vcdataObj.removeConstraint(nameOfConstraint)

Description

vcdataObj.removeConstraint(nameOfConstraint), removes the constraint from the variant configuration data object.

Input Arguments

nameOfConstraint

Name of the constraint to be removed.

Examples

% Define the variant configuration data object
vcdataObj = Simulink.VariantConfigurationData;

% Add a constraint named LinNotExtern
vcdataObj.addConstraint('LinNotExtern','((Ctrl==1)... || (PlantLocation ~=1))','Description of the constraint');

% Remove the constraint LinNotExtern
% from the variant configuration
vcdataObj.removeConstraint('LinNotExtern')
See Also
Simulink.VariantConfigurationData
Simulink.VariantConfigurationData.addConstraint
removeControlVariable

Class: Simulink.VariantConfigurationData
Package: Simulink

Remove a control variable from a variant configuration

Syntax

vcdataObj.removeControlVariable(nameOfConfiguration, nameOfControlVariable)

Description

vcdataObj.removeControlVariable(nameOfConfiguration, nameOfControlVariable) removes a control variable from a variant configuration.

Input Arguments

nameOfConfiguration
Name of the variant configuration.

nameOfControlVariable
Name of the control variable to be deleted.

Examples

% Define the variant configuration data object
vcdataObj = Simulink.VariantConfigurationData;

% Add a variant configuration named LinInterExp
vcdataObj.addConfiguration('LinInterExp',
    'Linear Internal Experimental Plant Controller');
% Add control variables SmartSensor1Mod and PlanLocation
vcdataObj.addControlVariables('LinInterExp', ...
    [struct('Name','SmartSensor1Mod','Value','2')]);

% Remove the control variable SmartSensor1Mod
% from the configuration LinInterExp
vcdataObj.removeControlVariable('LinInterExp', ...
    'SmartSensor1Mod')

See Also
Simulink.VariantConfigurationData|
Simulink.VariantConfigurationData.addControlVariables
**removeSubModelConfiguration**

**Class:** Simulink.VariantConfigurationData  
**Package:** Simulink

Remove from a variant configuration, the configuration to be used for a submodel.

**Syntax**

```
vcdataObj.removeSubModelConfiguration(nameOfConfiguration, nameOfSubModel)
```

**Description**

```
vcedataObj.removeSubModelConfiguration(nameOfConfiguration, nameOfSubModel), removes the configuration specified for a submodel.
```

**Input Arguments**

- **nameOfConfiguration**
  
  Name of the submodel configuration to be removed.

- **nameOfSubModel**
  
  Name of the submodel from which the configuration must be removed.

**Examples**

1. Add the path to the model file.
   ```
   addpath(fullfile(docroot,'toolbox','simulink','examples'));
   ```

2. Load the model.
   ```
   load_system('slexVariantManagementExample');
   ```
3  Add a new submodel configuration to LinInterExpNoNoise.

    vcdataObj.addSubModelConfigurations('LinInterExpNoNoise',...
    [struct('ModelName','slexVariantManagementExternalPlantMdlRef',...
    'ConfigurationName', 'LowFid')]);

4  Remove the submodel configuration LinInterExpNoNoise from the submodel
    slexVariantManagementExternalPlantMdlRef.

    vcdataObj.removeSubModelConfiguration('LinInterExpNoNoise',...
    'slexVariantManagementExternalPlantMdlRef')

See Also
Simulink.VariantConfigurationData|
Simulink.VariantConfigurationData.addSubModelConfigurations
setDefaultConfigurationName

Class: Simulink.VariantConfigurationData
Package: Simulink

Set name of the default variant configuration for a variant configuration data object

Syntax

vcdataObj.setDefaultConfigurationName(nameOfConfiguration)

Description

vcdataObj.setDefaultConfigurationName(nameOfConfiguration) sets the default configuration name. A variant configuration must exist with the same name. If an empty value is passed, then the default configuration name is cleared.

Input Arguments

nameOfConfiguration

Name of the configuration to be set as the default.

Examples

% Define the variant configuration data object
vcdataObj = Simulink.VariantConfigurationData;

% Add the LinInterExp variant configuration
vcdataObj.addConfiguration('LinInterExp',... 'Linear Internal Experimental Plant Controller');

% Set the configuration LinInterExp as default
vcdataObj.setDefaultConfigurationName('LinInterExp');
% Obtain the default variant configuration
dconfig = vcdataObj.getDefaultConfiguration

See Also
Simulink.VariantConfigurationData|
Simulink.VariantConfigurationData.getDefaultConfiguration
validateModel

Class: Simulink.VariantConfigurationData
Package: Simulink

Validate all variant blocks in the model and submodels in the hierarchy during simulation

Syntax

Simulink.VariantConfigurationData.validateModel(modelName)
Simulink.VariantConfigurationData.validateModel(modelName, configName)

Description

Simulink.VariantConfigurationData.validateModel(modelName), validates the model and referenced models during simulation.

Simulink.VariantConfigurationData.validateModel(modelName, configName), validates the model and referenced models during simulation optionally using a variant configuration.

Input Arguments

modelName

Name of the model

configName

Name of the configuration to be validated
Examples

1  Add the path to the model file.
   addpath(fullfile(docroot,'toolbox','simulink','examples'));
2  Load the model.
   load_system('slexVariantManagementExample');
3  Validate the model slexVariantManagementExample using the configuration LinInterExpNoNoise.
   [valid, errors] = Simulink.VariantConfigurationData.validateModel...
                    ('slexVariantManagementExample','LinInterExpNoNoise')

See Also
   Simulink.VariantConfigurationData|
   Simulink.VariantConfigurationData.existsFor|
   Simulink.VariantConfigurationData.getFor|
   Simulink.VariantConfigurationData.getOrCreateFor
VariantConfigurationData

Class: Simulink.VariantConfigurationData
Package: Simulink

Object constructor with optional arguments for variant configurations, constraints, and default configuration name.

Syntax

vardataObj = Simulink.VariantConfigurationData(
variantConfigurations)

Description

vardataObj = Simulink.VariantConfigurationData(
variantConfigurations), constructor that creates an empty variant configuration data object. Optionally, can also accept constraints and a default configuration name as inputs.

Input Arguments

variantConfigurations

Configurations that are part of the variant configuration data object.

constraints

Constraints to be satisfied by the model.

defaultConfigurationName

Name of the default configuration.
Examples

% Create an empty variant configuration data object
vcdataObj = Simulink.VariantConfigurationData

See Also
Simulink.VariantConfigurationData|
Simulink.VariantConfigurationData.addConfiguration|
Simulink.VariantConfigurationData.addConstraint|
Simulink.VariantConfigurationData.addControlVariables|
Simulink.VariantConfigurationData.addSubModelConfigurations
Simulink.VariantManager class

**Package:** Simulink

Class representing a set of Variant Manager functionality

**Description**

The variant manager class provides a set of methods to access Variant Manager functionality from the MATLAB command-line. Use an instance of Simulink.VariantManager class to:

- Convert the Subsystems or Model block to a Variant Subsystem.
- Find variables used in Variant control expressions.
- Generate a reduced model for specified variant configurations.
- Display or control behavior of a variant condition legend.

**Method**

- `variantLegend` Display or control behavior of variant condition legend
- `reduceModel` Generate reduced model for specified variant configurations
- `convertToVariant` Convert Subsystem, or Model block, or Variant Model block to a Variant Subsystem block

**See Also**

**Topics**

“Variant Manager Overview”
Simulink.VariantManager.variantLegend
Simulink.VariantManager.convertToVariant
variantLegend

Class: Simulink.VariantManager
Package: Simulink

Display or control behavior of variant condition legend

Syntax

Simulink.VariantManager.variantLegend(modelName, action)

Description

Simulink.VariantManager.variantLegend(modelName, action) displays or performs a specified action on the variant condition legend.

Input Arguments

modelName — Model for which the variant legend is displayed
character vector

Model for which the variant legend is displayed, specified as a character vector.

action — Task to be performed on the variant legend
'd open' | 'print' | 'showCodeConditions' | 'close'

Task to be performed on the variant condition legend for the model. You can specify the task as:

- 'open' — Displays the variant condition legend for a model. The model must be open. If the legend is opened for the first time, the model is updated.
- 'print' — Prints the data in the variant condition legend. The legend must be open. There is no preview before printing the legend.
• 'showCodeConditions' — Displays code generation conditions column in the variant condition legend. The legend must be open. showCodeConditions is used as a name-value pair and accepts 'on' or 'off' as its values.

• 'close' — Closes the variant condition legend belonging to the specified model.

Examples

```matlab
model = 'sldemo_variant_subsystems';
open_system(model);
% Open the variant condition legend
Simulink.VariantManager.variantLegend(model,'open');
% Display the code generation conditions
Simulink.VariantManager.variantLegend(model,'showCodeConditions','on');
```

See Also

Variant Subsystem

Topics

“Create a Simple Variant Model”
“What Are Variants and When to Use Them”

Introduced in R2017b
reduceModel

Class: Simulink.VariantManager
Package: Simulink

Generate reduced model for specified variant configurations

Syntax

Simulink.VariantManager.reduceModel(Model)
Simulink.VariantManager.reduceModel(Model, Name, Value)

Description

Simulink.VariantManager.reduceModel(Model) creates a reduced model for the specified configuration. The referenced models and library blocks are also reduced. By default, the name of the reduced model and any reduced child referenced model name is the original model name suffixed with _r.

Simulink.VariantManager.reduceModel(Model, Name, Value) specifies the reduction parameters in the Name and Value arguments form.

Input Arguments

Model — Model to be reduced
character vector

Required field. Model to be reduced, specified as a character vector.

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside single quotes ("'") and is caseinsensitive whereas, the value string is casesensitive. You can
specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

**NamedConfigurations — Variant configuration name**

```
{""} | character vector | 'cell array'
```

Specifies the names of variant configurations. By default, current values of variant control variables are used for reduction.

**VariableGroups — Variant control variable value**

```
{""} | 'cell array'
```

Specifies the variant control variable values to be used for reduction. By default, the current values of variant control variables are used.

The specified values must be a cell array with variant control variable names and their corresponding values.

**Note** 'VariableGroups' and 'NamedConfigurations' are mutually exclusive.

Consider this example:

```matlab
Simulink.VariantManager.reduceModel('iv_model', ... 'VariableGroups', ... {{'V',1,'W',1},{'V',2,'W',2}})
```

**FullRangeVariables — Variant control variable value**

```
{""} | 'cell array'
```

Specifies the full-range variant control variable values to be used for reduction. This allows you to reduce a model for all valid values of the specified variant control variable. Provide a reference value for variant control variable that results in a successful model compilation.

Consider this example:

```matlab
Simulink.VariantManager.reduceModel('slexVariantReducer',... 'VariableGroups',{'}V',1},... 'FullRangeVariables',{'}W',1});
```
You can specify a variant control variable, 'W', as a full-range variant control variable. This allows you to reduce the model for all valid values of variable 'W'. In the example, full-range variant control variable W uses a reference value of 1.

**OutputFolder** — Folder to store the reduced model
character vector

Specifies the folder to place the reduced models and related artifacts. By default, the reduced models are generated in .\reducedModel subfolder in the original model folder.

**PreserveSignalAttributes** — Preserve the signal attributes in the reduced model
{true} | false

When the value is true, the Variant Reducer preserves the compiled signal attributes between the original and reduced models by adding signal specification blocks at appropriate block ports in the reduced model. Compiled signal attributes include signal data types, signal dimensions, compiled sample times, etc.

**Verbose** — Displays step details
true | {false}

When the value is true, the Variant Reducer displays details of the steps performed during model reduction.

**ModelSuffix** — Reduced model name suffix
{_r}

Specifies the suffix to append to the reduced models and the related artifacts.

**GenerateSummary** — Generates a summary html file
true | {false}

When the value is true, the Variant Reducer generates a html file with details about the reduced model and any modifications that may be required for masks and callbacks.

**Note** To generate summary, you must have Simulink Report Generator license.
Examples

% Reduce model based on its variant control variable values in the base workspace.
Simulink.VariantManager.reduceModel('sldemo_variant_subsystems');

% Reduce the model associated with a variant configuration data object and configurations to be retained in
Simulink.VariantManager.reduceModel('slexVariantManagementExample', ...
  'NamedConfigurations', {'LinInterStd', 'NonLinExterHighFid'});

% Reduce the model by specifying variant control variable values. Here, two groups are specified corresponding to
% {V==1, W==1}, and {V==2, W==2} respectively.
Simulink.VariantManager.reduceModel('iv_model', ...
  'VariableGroups', ...
  {{'V',1,'W',1},{'V',2,'W',2}});

% Reduce the model by specifying variant control variable values where 'W' is a full-range variant control variable.
% automatically maps the specification to correspond to the following four explicit groups: {V==1, W==1}, {V==1, W==2},
% {V==1, W==3}, and {V==1, W==0} respectively.
Simulink.VariantManager.reduceModel('slexVariantReducer', ...
  'VariableGroups', ...
  {'V',1},
  'FullRangeVariables',{'W',1});

See Also

Variant Subsystem

Topics

“Create a Simple Variant Model”
“What Are Variants and When to Use Them”
Simulink.VariantManager
Simulink.VariantManager.variantLegend

Introduced in R2016a
convertToVariant

**Class:** Simulink.VariantManager  
**Package:** Simulink

Convert Subsystem, or Model block, or Variant Model block to a Variant Subsystem block

**Syntax**

```matlab
variant_subsystem = Simulink.VariantManager.convertToVariant(block)
variant_subsystem = Simulink.VariantManager.convertToVariant(blockHandle)
```

**Description**

`variant_subsystem = Simulink.VariantManager.convertToVariant(block)` or `variant_subsystem = Simulink.VariantManager.convertToVariant(blockHandle)` converts a Subsystem, or Model block, or Variant Model block to a Variant Subsystem block. A Variant Subsystem can contain Subsystems, Model blocks, or both as choices.

Consider this model with Subsystem block.

![Subsystem diagram](image)

You can convert this Subsystem block to a Variant Subsystem block using the `convertToVariant` method.
Similarly you can convert a Variant Model block to a Variant Subsystem block.

To convert a Variant Model block to a Variant Subsystem block, you can also use the Upgrade Advisor check, **Identify Variant Model blocks and convert those to Variant Subsystem containing Model block choices**. This check provides a **Fix** button to convert Variant Model blocks to Variant Subsystem blocks. For more information on using Upgrade Advisor check to convert a Variant Model block to a Variant Subsystem block, see “Upgrade Advisor Checks”

If there are inconsistencies in the port name or port number in models referenced by Variant Model block, Simulink corrects these inconsistencies while converting the Variant Model block to Variant Subsystem block.

Consider this model with Variant Model block having different port names.

When you convert this Variant Model block to a Variant Subsystem block, the inconsistencies are corrected automatically.
**Note** Future releases will no longer support using a Model block to contain model variants. You can use the `convertToVariant` method to convert model variants so that Model blocks are contained in a Variant Subsystem. Use of a Variant Subsystem block provides these advantages:

- Allows you to mix Model and Subsystem blocks as variant choices
- Supports flexible I/O, so that all variants do not need to have the same number of input and output ports

For an example of a model that uses a Variant Subsystem block as a container for variant models, see “Model Reference Variants” on page 15-605.

**Limitations**

You cannot convert a Subsystem block that meets the following condition:

- The Subsystem block is in a Simscape model that has Editing Mode set to **Restricted**.
- The Variant Model block has a mix of control ports or there is a name mismatch in control port types.
- The Variant Model block has control ports with different port numbers and name mapping.
Input Arguments

block — Subsystem or Model block to convert
block path | block handle

The path or block handle of the Subsystem or Model block to convert to a Variant Subsystem block. Specify a block path as a character vector and a block handle as a scalar.

Output Arguments

variant_subsystem — Handle of Variant Subsystem block
scalar returned by Simulink

If you specify an output argument, the method returns the block handle of the Variant Subsystem block created by the conversion.

Examples

```matlab
open_system('sldemo_mdlref_conversion');
    Simulink.VariantManager.convertToVariant('sldemo_mdlref_conversion/Bus Counter');
```

Alternative Functionality

Simulink Editor

In the Simulink Editor, right-click the Model block and select Subsystems & Model Reference > Convert to > Variant Subsystem.

See Also

Variant Subsystem

Topics

“Create a Simple Variant Model”
“What Are Variants and When to Use Them”

Introduced in R2017b
Simulink.WorkspaceVar

Store information about workspace variables and blocks that use them

**Note** Simulink.WorkspaceVar is not recommended. Use Simulink.VariableUsage instead.

**Description**

A Simulink.WorkspaceVar object describes attributes of a workspace variable and lists the blocks that use the variable.

**Creation**

The Simulink.findVars function returns one or more Simulink.WorkspaceVar objects that embody the results of searching for variables.

Only the Simulink.WorkspaceVar function can set any field value in a Simulink.WorkspaceVar object. The fields are otherwise read-only.

**Syntax**

```matlab
varObj = Simulink.WorkspaceVar(varNames,wkspName)
```

**Description**

`varObj = Simulink.WorkspaceVar(varNames,wkspName)` creates an array of Simulink.WorkspaceVar objects to describe the variables `varNames`. The constructor sets the `Name` property of each object to one of the variable names specified by `varNames`, and sets the `Workspace` property of all the objects to the workspace specified by `wkspName`. You can specify `varNames` with variables that are not used in any loaded models.
**Input Arguments**

**varNames — Names of target variables**
character vector | cell array of character vectors

Names of target variables, specified as a character vector or a cell array of character vectors. The constructor creates a Simulink.WorkspaceVar object for each variable name. You can specify `varNames` with variables that are not used in any loaded models.

Example: 'k'
Example: {'k','asdf','fuelFlow'}
Data Types: char | cell

**wkspName — Name of containing workspace**
character vector

Name of the workspace that defines the target variables, specified as a character vector. For example, you can specify the MATLAB base workspace. The constructor also determines and sets the WorkspaceType property of each of the returned Simulink.WorkspaceVar objects.

Example: 'base workspace'
Example: 'myModel'
Example: 'myDictionary.sldd'
Data Types: char

**Properties**

**Name — Name of variable**
' ' (empty character vector) (default) | character vector

This property is read-only.

Name of the variable described by the object, returned as a character vector.

**Workspace — Name of workspace that contains variable**
' ' (empty character vector (default) | character vector

This property is read-only.
Name of the workspace that contains the variable, returned as a character vector. For example:

<table>
<thead>
<tr>
<th>Workspace value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>'base workspace'</td>
<td>The MATLAB base workspace</td>
</tr>
<tr>
<td>'MyModel'</td>
<td>The model workspace for the model MyModel.</td>
</tr>
<tr>
<td>'MyModel/Mask1'</td>
<td>The mask workspace for the masked block Mask1 in the model MyModel.</td>
</tr>
</tbody>
</table>

**WorkspaceType — Type of workspace containing variable**

'unknown' (default) | 'base' | 'model' | 'mask'  

This property is read-only.

Type of workspace that contains the variable, returned as a character vector. The possible values are:

- 'base' — The base workspace
- 'model' — A model workspace
- 'mask' — A mask workspace

**UsedByBlocks — Users of variable**

{} (empty cell array) (default) | cell array of character vectors

This property is read-only.

Users of the variable, returned as a cell array of character vectors. Each character vector identifies a block that uses the variable. The Simulink.findVars function populates this property.

**Object Functions**

- `intersect` Return intersection of two arrays of Simulink.VariableUsage objects
- `setdiff` Return difference between two arrays of Simulink.VariableUsage objects

**Examples**
Create Object That Represents Variable in Base Workspace

Return a Simulink.WorkspaceVar object for a variable k in the base workspace.

```matlab
var = Simulink.WorkspaceVar('k', 'base workspace');
```

Represent All Variables in the Base Workspace

Return an array of Simulink.WorkspaceVar objects containing one object for each variable returned by the who command in the base workspace.

```matlab
vars = Simulink.WorkspaceVar(who,WkspName)
```

Represent All Variables in a Model Workspace

Return an array of Simulink.WorkspaceVar objects that describes all the variables in a model workspace.

```matlab
hws = get_param('mymodel', 'ModelWorkspace');
vars = Simulink.WorkspaceVar(hws.whos, 'MyModel')
```

Represent All Variables in a Mask Workspace

Return an array of Simulink.WorkspaceVar objects that describes all the variables in a mask workspace.

```matlab
maskVars = get_param('mymodel/maskblock', 'MaskWSVariables');
vars = Simulink.WorkspaceVar(maskVars, 'mymodel/maskblock');
```

See Also

Simulink.findVars | intersect | setdiff

Introduced in R2010a
Simulink.VariableUsage

Store information about the relationship between variables and blocks in models

Description

A Simulink.VariableUsage object describes where a variable is used in models.

Use this information to:

• Prepare to permanently store the variables in files and workspaces. For more information about storing variables for a model, see “Determine Where to Store Variables and Objects for Simulink Models”.
• Reduce the number of variables that you need to store by eliminating unused variables.
• Prepare to partition variables and establish variable ownership when you work in a team.

To analyze variable usage in models, use Simulink.VariableUsage objects together with the Simulink.findVars function. The function returns and accepts Simulink.VariableUsage objects as arguments. For more information, see Simulink.findVars.

A Simulink.VariableUsage object can also describe the usage of an enumerated data type.

Only a Simulink.VariableUsage constructor or the Simulink.findVars function can set property values in a Simulink.VariableUsage object. The properties are otherwise read only.

Creation

The Simulink.findVars function returns Simulink.VariableUsage objects.

To create variable usage objects for use as a filter when using Simulink.findVars, use the Simulink.VariableUsage function described below.
Syntax

```
variableUsageObj = Simulink.VariableUsage(varNames,sourceName)
```

Description

```
variableUsageObj = Simulink.VariableUsage(varNames,sourceName) creates
an array of Simulink.VariableUsage objects to describe the variables varNames. The
constructor sets the Name property of each object to one of the variable names specified
by varNames, and sets the Source property of all the objects to the source specified by
sourceName. You can specify varNames with variables that are not used in any loaded
models.
```

Input Arguments

```
varNames — Names of target variables
character vector | cell array of character vectors

Names of target variables, specified as a character vector or a cell array of character
vectors. The constructor creates a Simulink.VariableUsage object for each variable
name.
```

Example: 'k'
Example: {'k','asdf','fuelFlow'}
Data Types: char | cell

```
sourceName — Name of variable source
character vector

Name of the source that defines the target variables, specified as a character vector. For
example, you can specify the MATLAB base workspace or a data dictionary as a source.
The constructor also determines and sets the SourceType property of each of the
returned Simulink.VariableUsage objects.
```

Example: 'base workspace'
Example: 'myModel'
Example: 'myDictionary.sldd'
Data Types: char
Properties

Name — Name of variable or enumerated type
'' (empty character vector) (default) | character vector

This property is read-only.

The name of the variable or enumerated data type the object describes, returned as a character vector.

Source — Name of defining workspace
'' (empty character vector) (default) | character vector

This property is read-only.

The name of the workspace or data dictionary that defines the described variable, returned as a character vector. The table shows some examples.

<table>
<thead>
<tr>
<th>Source Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>'base workspace'</td>
<td>MATLAB base workspace</td>
</tr>
<tr>
<td>'MyModel'</td>
<td>Model workspace for the model MyModel</td>
</tr>
<tr>
<td>'MyModel/Mask1'</td>
<td>Mask workspace for the masked block Mask1 in the model MyModel</td>
</tr>
<tr>
<td>'sldemo_fuelsys_dd_controller.sldd'</td>
<td>The data dictionary named 'sldemo_fuelsys_dd_controller.sldd'</td>
</tr>
</tbody>
</table>

The table shows some examples if you created the Simulink.VariableUsage object by using the Simulink.findVars function to find enumerated data types.

<table>
<thead>
<tr>
<th>Source Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>'BasicColors.m'</td>
<td>The enumerated type is defined in the MATLAB file 'BasicColors.m'.</td>
</tr>
<tr>
<td>''</td>
<td>The enumerated type is defined dynamically and has no source.</td>
</tr>
<tr>
<td>'sldemo_fuelsys_dd_controller.sldd'</td>
<td>The enumerated type is defined in the data dictionary named 'sldemo_fuelsys_dd_controller.sldd'.</td>
</tr>
</tbody>
</table>
**SourceType — Type of defining workspace**

'unknown source' (default) | 'base workspace' | 'model workspace' | 'mask workspace' | 'data dictionary'

This property is read-only.

The type of the workspace that defines the variable, returned as a character vector. The possible values are:

- 'base workspace'
- 'model workspace'
- 'mask workspace'
- 'data dictionary'

If you created the `Simulink.VariableUsage` object by using the `Simulink.findVars` function to find enumerated data types, the possible values are:

- 'MATLAB file'
- 'dynamic class'
- 'data dictionary'

**Users — Blocks that use the variable or models that use the enumerated type**

{} (empty cell array) (default) | cell array of character vectors

This property is read-only.

Blocks that use the variable or models that use the enumerated type, returned as a cell array of character vectors. Each character vector names a block or model that uses the variable or enumerated type. The `Simulink.findVars` function populates this property.

**Object Functions**

- `intersect` Return intersection of two arrays of `Simulink.VariableUsage` objects
- `setdiff` Return difference between two arrays of `Simulink.VariableUsage` objects
- `union` Return union of two arrays of `Simulink.VariableUsage` objects

**Examples**
Create Object That Represents Variable in Base Workspace

Return a Simulink.VariableUsage object for a variable k in the base workspace.

```matlab
var = Simulink.VariableUsage('k','base workspace');
```

You can use var as a filter for the Simulink.findVars function.

Represent All Variables in the Base Workspace

Return an array of Simulink.VariableUsage objects containing one object for each variable returned by the whos command in the base workspace.

```matlab
vars = Simulink.VariableUsage(whos,'base workspace')
```

Represent All Variables in a Model Workspace

Return an array of Simulink.VariableUsage objects that describes all the variables in a model workspace.

```matlab
hws = get_param('mymodel','ModelWorkspace');
vars = Simulink.VariableUsage(hws.whos,'MyModel')
```

Represent All Variables in a Mask Workspace

Return an array of Simulink.VariableUsage objects that describes all the variables in a mask workspace.

```matlab
maskVars = get_param('mymodel/maskblock','MaskWSVariables');
vars = Simulink.VariableUsage(maskVars,'mymodel/maskblock');
```

See Also
Simulink.data.existsInGlobal | Simulink.findVars

Topics
“Manage Design Data”
Introduced in R2012b
Simulink.VariantConfigurationAnalysis class

Package: Simulink

Class representing a set of Variant Configuration Analysis functionality

Description

The Simulink.VariantConfigurationAnalysis class has methods that enable you to analyze or compare model references and libraries for each variant configuration in a model containing variant blocks. Use an instance of Simulink.VariantConfigurationAnalysis class to do the following:

• Get a list of active blocks for variant configuration
• Find blocks which are active across all variant configurations
• Find blocks which are inactive in all variant configurations
• Get a list of blocks which differ in active choice between variant configurations
• Get a list of active models for variant configuration
• Get a list of libraries which are used in variant configuration
• View variant configuration analysis in user interface screen
• Hide variant configuration analysis report
• View variant condition for blocks in given variant configuration

Properties

NamedConfigurations

Specifies the names of variant configuration(s) you want to analyze. The names are defined by the Variant Manager and saved in a variant configuration object associated with the model. If the model is associated with a variant configuration object, then the VariantConfigurationObject parameter is set to a Simulink.VariantConfigurationData object. For example, consider a variant configuration object "vcd", then NamedConfigurations must be one or multiple
configuration names that is defined in "vcd". NamedConfigurations can be a string or a cell array of strings.

**Method**

- `getActiveBlocks` Returns a list of active blocks for variant configuration
- `getAlwaysActiveBlocks` Displays a list of blocks that are always active across all variant configurations
- `getNeverActiveBlocks` Returns a list of blocks that are inactive in all variant configurations
- `getBlockDifferences` Returns a list of blocks that differ in active choice between variant configurations
- `getDependentModels` Returns a list of active models for variant configurations
- `getDependentLibraries` Returns a list of libraries that are used in variant configurations
- `showUI` Displays variant configuration analysis in user interface view
- `hideUI` Hides variant configuration analysis report
- `getVariantCondition` Returns variant conditions for blocks in given variant configuration

**See Also**

**Topics**

“Variant Manager Overview”
getActiveBlocks

Class: Simulink.VariantConfigurationAnalysis
Package: Simulink

Returns a list of active blocks for variant configuration

Syntax

activeBlocks = VarConfigObj.getActiveBlocks(ConfigName)

Description

activeBlocks = VarConfigObj.getActiveBlocks(ConfigName) returns a list of active blocks for variant configuration.

Input Arguments

VarConfigObj — Variant configuration object
character vector

Variant configuration object for a specified configuration. You can use the Simulink.VariantConfigurationAnalysis method to obtain this value.

ConfigName — Variant configuration name
character vector | string

Specifies the name of variant configuration.

Output Arguments

activeBlocks — List of active blocks
cell array

Returns a list of active blocks as a cell array.
Examples

% Open model
open_system('slexVariantManagementExample');

% Get variant configuration object
VarConfigObj = Simulink.VariantConfigurationAnalysis('slexVariantManagementExample', 'NamedConfigurations', ...
'NonLinExterLowFid', 'NonLinExterHighFid'))

% Get active blocks
ActiveBlocks = VarConfigObj.getActiveBlocks('LinInterStd')

See Also

“Analyze Variant Configurations in Models Containing Variant Blocks” | “Variant Manager Overview”

Topics

“Create a Simple Variant Model”

Introduced in R2019a
**getAlwaysActiveBlocks**

**Class:** Simulink.VariantConfigurationAnalysis  
**Package:** Simulink

Displays a list of blocks that are always active across all variant configurations

**Syntax**

```
alwaysActiveBlocks = VarConfigObj.getAlwaysActiveBlocks()
```

**Description**

```
alwaysActiveBlocks = VarConfigObj.getAlwaysActiveBlocks() displays a list of blocks that are always active across all variant configurations.
```

**Input Arguments**

VarConfigObj — Variant configuration object  
character vector

Variant configuration object for a specified configuration. You can use the Simulink.VariantConfigurationAnalysis method to obtain this value.

**Output Arguments**

alwaysActiveBlocks — List of always active blocks  
cell array

Displays a list of blocks that are always active, as a cell array.
Examples

% Open model
open_system('slexVariantManagementExample');

% Get variant configuration object
VarConfigObj = Simulink.VariantConfigurationAnalysis('slexVariantManagementExample', 'NamedConfigurations',
'NonLinExterLowFid', 'NonLinExterHighFid'))

% Get always active blocks
alwaysActiveBlocks = VarConfigObj.getAlwaysActiveBlocks()

See Also
“Analyze Variant Configurations in Models Containing Variant Blocks” | “Variant Manager Overview”

Topics
“Create a Simple Variant Model”

Introduced in R2019a
**getNeverActiveBlocks**

**Class:** Simulink.VariantConfigurationAnalysis  
**Package:** Simulink

Returns a list of blocks that are inactive in all variant configurations

**Syntax**

`getNeverActiveBlocks = VarConfigObj.getNeverActiveBlocks()`  

**Description**

`getNeverActiveBlocks = VarConfigObj.getNeverActiveBlocks()` returns a list of blocks that are inactive in all variant configurations.

**Input Arguments**

`VarConfigObj` — Variant configuration object  
character vector

Variant configuration object for a specified configuration. You can use the `Simulink.VariantConfigurationAnalysis` method to obtain this value.

**Output Arguments**

`neverActiveBlocks` — List of never active blocks  
cell array

Returns a list of blocks that are never active, as a cell array.
Examples

% Open model
open_system('slexVariantManagementExample');

% Get variant configuration object
VarConfigObj = Simulink.VariantConfigurationAnalysis('slexVariantManagementExample', 'NamedConfigurations', ...
'NonLinExterLowFid', 'NonLinExterHighFid'))

% Get never active blocks
neverActiveBlocks = VarConfigObj.getNeverActiveBlocks()

See Also

“Analyze Variant Configurations in Models Containing Variant Blocks” | “Variant Manager Overview”

Topics

“Create a Simple Variant Model”

Introduced in R2019a
**getBlockDifferences**

*Class:* Simulink.VariantConfigurationAnalysis  
*Package:* Simulink

Returns a list of blocks that differ in active choice between variant configurations.

**Syntax**

```matlab
getBlockDifferences = VarConfigObj.getBlockDifferences()
```

**Description**

`getBlockDifferences = VarConfigObj.getBlockDifferences()` returns a list of blocks that are active in some of the analyzed configurations and inactive in other configurations.

**Input Arguments**

*VarConfigObj* — Variant configuration object  
character vector

Variant configuration object for a specified configuration. You can use the `Simulink.VariantConfigurationAnalysis` method to obtain this value.

**Output Arguments**

*getBlockDifferences* — List of blocks that differ in active choice  
cell array

Returns a list of blocks that differ in active choice between variant configurations, as a cell array.
Examples

% Open model
open_system('slexVariantManagementExample');

% Get variant configuration object
VarConfigObj = Simulink.VariantConfigurationAnalysis('slexVariantManagementExample', 'NamedConfigurations', ...
'NonLinExterLowFid', 'NonLinExterHighFid'))

% Get block differences
getBlockDifferences = VarConfigObj.getBlockDifferences()

See Also

“Analyze Variant Configurations in Models Containing Variant Blocks” | “Variant Manager Overview”

Topics

“Create a Simple Variant Model”

Introduced in R2019a
getDependentModels

**Class:** Simulink.VariantConfigurationAnalysis  
**Package:** Simulink

Returns a list of active models for variant configurations

**Syntax**

```matlab
dependentModels = VarConfigObj.getDependentModels(ConfigName)
```

**Description**

```matlab
dependentModels = VarConfigObj.getDependentModels(ConfigName) returns a list of active models for variant configuration.
```

**Input Arguments**

- **VarConfigObj** — Variant configuration object  
  character vector
  
  Variant configuration object for a specified configuration. You can use the Simulink.VariantConfigurationAnalysis method to obtain this value.

- **ConfigName** — Variant configuration name  
  character vector  
  string
  
  Specifies the name of variant configuration.

**Output Arguments**

- **dependentModels** — List of models  
  cell array
  
  Returns a list of active models, as a cell array.
Examples

% Open model
open_system('slexVariantManagementExample');

% Get variant configuration object
VarConfigObj = Simulink.VariantConfigurationAnalysis('slexVariantManagementExample', 'NamedConfigurations',
    'NonLinExterLowFid', 'NonLinExterHighFid'))

% Get dependent models
dependentModels = VarConfigObj.getDependentModels('NonLinExterLowFid')

See Also
“Analyze Variant Configurations in Models Containing Variant Blocks” | “Variant Manager Overview”

Topics
“Create a Simple Variant Model”

Introduced in R2019a
getDependentLibraries

Class: Simulink.VariantConfigurationAnalysis
Package: Simulink

Returns a list of libraries that are used in variant configurations

Syntax

dependentLibraries = VarConfigObj.getDependentLibraries(ConfigName)

Description

dependentLibraries = VarConfigObj.getDependentLibraries(ConfigName) returns a list of libraries that are used in variant configuration.

Input Arguments

VarConfigObj — Variant configuration object
character vector

Variant configuration object for a specified configuration. You can use the Simulink.VariantConfigurationAnalysis method to obtain this value.

ConfigName — Variant configuration name
character vector | | string

Specifies the name of variant configuration.

Output Arguments

dependentLibraries — List of libraries
cell array

Returns a list of libraries, as a cell array.
Examples

% Open model
open_system('slexVariantManagementExample');

% Get variant configuration object
VarConfigObj = Simulink.VariantConfigurationAnalysis('slexVariantManagementExample', 'NamedConfigurations',
'NonLinExterLowFid', 'NonLinExterHighFid'))

% Get dependent libraries
dependentLibraries = VarConfigObj.getDependentLibraries('LinInterStd')

See Also
“Analyze Variant Configurations in Models Containing Variant Blocks” | “Variant Manager Overview”

Topics
“Create a Simple Variant Model”

Introduced in R2019a
showUI

Class: Simulink.VariantConfigurationAnalysis
Package: Simulink

Displays variant configuration analysis in user interface view

Syntax

VarConfigObj.showUI()

Description

VarConfigObj.showUI(), displays variant configuration analysis in user interface view.

Examples

% Open model
open_system('slexVariantManagementExample');

% Get variant configuration object
VarConfigObj = Simulink.VariantConfigurationAnalysis('slexVariantManagementExample', 'NamedConfigurations', {'LinInterStd', 'NonLinExterLowFid', 'NonLinExterHighFid'})

% Show UI
VarConfigObj.showUI()

See Also

“Analyze Variant Configurations in Models Containing Variant Blocks” | “Variant Manager Overview”

Topics

“Create a Simple Variant Model”

Introduced in R2019a
hideUI

Class: Simulink.VariantConfigurationAnalysis
Package: Simulink

Hides variant configuration analysis report

Syntax

VarConfigObj.hideUI()

Description

VarConfigObj.hideUI(), hides the variant configuration analysis report.

Examples

% Open model
open_system('slexVariantManagementExample');

% Get variant configuration object
VarConfigObj = Simulink.VariantConfigurationAnalysis('slexVariantManagementExample', 'NamedConfigurations', 'NonLinExterLowFid', 'NonLinExterHighFid')

% Hide UI
VarConfigObj.hideUI()

See Also

“Analyze Variant Configurations in Models Containing Variant Blocks” | “Variant Manager Overview”

Topics

“Create a Simple Variant Model”

Introduced in R2019b
getVariantCondition

Class: Simulink.VariantConfigurationAnalysis
Package: Simulink

Returns variant conditions for blocks in given variant configuration

Syntax

```
variantCondition = VarConfigObj.getVariantCondition(ConfigName)
```

Description

```
variantCondition = VarConfigObj.getVariantCondition(ConfigName),
```
returns the variant condition for block in given variant configuration.

Input Arguments

- **VarConfigObj** — Variant configuration object
  character vector

Variant configuration object for a specified configuration. You can use the Simulink.VariantConfigurationAnalysis method to obtain this value.

- **ConfigName** — Variant configuration name
  character vector | string

  Specifies the name of variant configuration.

Output Arguments

- **variantCondition** — Variant conditions for a block
  cell array

  Returns variant conditions for blocks in given variant configuration.
Examples

% Open model
open_system('slexVariantManagementExample');

% Get variant configuration object
VarConfigObj = Simulink.VariantConfigurationAnalysis('slexVariantManagementExample',... 
'NamedConfigurations', {'LinInterStd','NonLinExterLowFid','NonLinExterHighFid'})

% Get variant condition
VariantCondition = VarConfigObj.getVariantCondition('NonLinExterLowFid',... 
'slexVariantManagementExample/Controller/Linear Controller')

See Also

“Analyze Variant Configurations in Models Containing Variant Blocks” | “Variant Manager Overview”

Topics

“Create a Simple Variant Model”

Introduced in R2019b
**intersect**

**Package:** Simulink

Return intersection of two arrays of Simulink.VariableUsage objects

**Syntax**

\[ \text{VarsOut} = \text{intersect}(\text{VarsIn1}, \text{VarsIn2}) \]

**Description**

\[ \text{VarsOut} = \text{intersect}(\text{VarsIn1}, \text{VarsIn2}) \]

returns an array that identifies the variables described in \( \text{VarsIn1} \) and in \( \text{VarsIn2} \), which are arrays of Simulink.VariableUsage objects. If a variable is described by a Simulink.VariableUsage object in \( \text{VarsIn1} \) and in \( \text{VarsIn2} \), the function returns a Simulink.VariableUsage object that stores the variable usage information from both objects in the Users property.

**Examples**

**Compare Variables Used by Models**

Given two models, discover the variables needed by both models.
model1Vars = Simulink.findVars('model1');
model2Vars = Simulink.findVars('model2');
commonVars = intersect(model1Vars, model2Vars);

**Input Arguments**

**VarsIn1** — First array of variables for comparison
array of Simulink.VariableUsage objects

First array of variables for comparison, specified as an array of Simulink.VariableUsage objects.

**VarsIn2** — Second array of variables for comparison
array of Simulink.VariableUsage objects

Second array of variables for comparison, specified as an array of Simulink.VariableUsage objects.

**Output Arguments**

**VarsOut** — Variables described in both input arrays
array of Simulink.VariableUsage objects

Variables that are described in both input arrays, returned as an array of Simulink.VariableUsage objects. The function returns an object for each variable that is described in VarsIn1 and in VarsIn2.

**See Also**

Simulink.VariableUsage | Simulink.findVars | setdiff | union

**Topics**

“Manage Design Data”

**Introduced in R2012b**
setdiff

**Package:** Simulink

Return difference between two arrays of Simulink.VariableUsage objects

**Syntax**

$$\text{VarsOut} = \text{setdiff(} \text{VarsIn1}, \text{VarsIn2})$$

**Description**

$$\text{VarsOut} = \text{setdiff(} \text{VarsIn1}, \text{VarsIn2})$$ returns an array that identifies the variables described in \text{VarsIn1} but not in \text{VarsIn2}, which are arrays of Simulink.VariableUsage objects. If a variable is described by a Simulink.VariableUsage object in \text{VarsIn1} but not in \text{VarsIn2}, the function returns a copy of the object.

setdiff compares the Name, Source, and SourceType properties of the Simulink.VariableUsage objects in \text{VarsIn1} with the same properties of the objects in \text{VarsIn2}. If \text{VarsIn1} and \text{VarsIn2} each contain a Simulink.VariableUsage object with the same values for these three properties, the objects describe the same variable, and setdiff does not return an object to describe it.

To create Simulink.VariableUsage objects that describe the usage of variables in a model, use the Simulink.findVars function.

**Examples**

**Determine Variable Usage Difference Between Models**

Given two models, discover the variables that are needed by the first model but not the second model.
model1Vars = Simulink.findVars('model1');
model2Vars = Simulink.findVars('model2');
differentVars = setdiff(model1Vars,model2Vars);

**Find Variables Not Used by Model**

Locate all variables in the base workspace that are not used by a loaded model that has been recently compiled.

models = find_system('type','block_diagram','LibraryType','None');
base_vars = Simulink.VariableUsage(who,'base workspace');
used_vars = Simulink.findVars(models,'WorkspaceType','base');
unusedVars = setdiff(base_vars,used_vars);

**Input Arguments**

**VarsIn1** — First array of variables for comparison

array of Simulink.VariableUsage objects

First array of variables for comparison, specified as an array of Simulink.VariableUsage objects.

**VarsIn2** — Second array of variables for comparison

array of Simulink.VariableUsage objects

Second array of variables for comparison, specified as an array of Simulink.VariableUsage objects.

**Output Arguments**

**VarsOut** — Variables described in first array but not second array

array of Simulink.VariableUsage objects

Variables that are described in the first input array but not in the second input array, returned as an array of Simulink.VariableUsage objects. The function returns an object for each variable that is described in VarsIn1 but not in VarsIn2.
See Also
Simulink.VariableUsage | Simulink.findVars | intersect | union

Topics
“Manage Design Data”

Introduced in R2012b
union

Package: Simulink

Return union of two arrays of Simulink.VariableUsage objects

Syntax

$$\text{VarsOut} = \text{union(VarsIn1,VarsIn2)}$$

Description

$$\text{VarsOut} = \text{union(VarsIn1,VarsIn2)}$$ returns an array that is a combined set of the variables from VarsIn1 and VarsIn2, which are arrays of Simulink.VariableUsage objects. The output array does not contain duplicates. If a variable is described by a Simulink.VariableUsage object in VarsIn1 and in VarsIn2, the function returns a Simulink.VariableUsage object that stores the variable usage information from both objects in the Users property.

To create Simulink.VariableUsage objects that describe the usage of variables in a model, use the Simulink.findVars function.

Examples

Find the Union of Variables Used by Models

Given two models, discover the combined set of variables from both models.
model1Vars = Simulink.findVars('model1');
model2Vars = Simulink.findVars('model2');
unionVars = union(model1Vars, model2Vars);

Input Arguments

**VarsIn1 — First array of variables for union**
array of Simulink.VariableUsage objects

First array of variables for union, specified as an array of Simulink.VariableUsage objects.

**VarsIn2 — Second array of variables for union**
array of Simulink.VariableUsage objects

Second array of variables for union, specified as an array of Simulink.VariableUsage objects.

Output Arguments

**VarsOut — Combined set of variables described in input arrays**
array of Simulink.VariableUsage objects

Combined set of variables that are described in input arrays, returned as an array of Simulink.VariableUsage objects. The function returns an object for each variable that is described in VarsIn1 or in VarsIn2.

See Also

Simulink.VariableUsage | Simulink.findVars | intersect | setdiff

Topics

“Manage Design Data”

Introduced in R2019b
Simulink.data.Dictionary class

Package: Simulink.data

Configure data dictionary

Description

An object of the Simulink.data.Dictionary class represents a data dictionary. The object allows you to perform operations on the data dictionary such as save or discard changes, import data from the base workspace, and add other data dictionaries as references.

Construction

The functions Simulink.data.dictionary.create and Simulink.data.dictionary.open create a Simulink.data.Dictionary object.

Properties

DataSources — Referenced data dictionaries
cell array of character vectors

This property is read-only.

Referenced data dictionaries by file name, returned as a cell array of character vectors. This property only lists directly referenced dictionaries whose parent is the Simulink.data.Dictionary object.

EnableAccessToBaseWorkspace — Specify whether models can use design data in the base workspace
false (default) | true

Whether linked models can use design data in the base workspace, specified as true or false.
To determine whether a dictionary provides access to the base workspace (including through referenced dictionaries), query the `HasAccessToBaseWorkspace` property.

For more information about this property, including restrictions that limit your ability to interact with base workspace data through the dictionary, see “Continue to Use Shared Data in the Base Workspace”.

Data Types: logical

**HasAccessToBaseWorkspace** — Query whether models can use design data in the base workspace

0 (default) | 1

This property is read-only.

Query whether models can use design data in the base workspace, returned as 1 (true) or 0 (false). If the dictionary or a referenced dictionary has the `EnableAccessToBaseWorkspace` property set to `true`, this property returns 1.

Use this property to determine whether models that link to the dictionary can use design data in the base workspace. You do not need to query each referenced dictionary to determine whether it has the `EnableAccessToBaseWorkspace` property set to `true`.

Data Types: logical

**HasUnsavedChanges** — Indicator of unsaved changes

0 | 1

This property is read-only.

Indicator of unsaved changes to the data dictionary, returned as 0 or 1. The value is 1 if changes have been made since last data dictionary save and 0 if not.

**NumberOfEntries** — Total number of entries in data dictionary

integer

This property is read-only.

Total number of entries in data dictionary, including those in referenced dictionaries, returned as an integer.
Methods

addDataSource  Add reference data dictionary to parent data dictionary
close          Close connection between data dictionary and
               Simulink.data.Dictionary object
discardChanges Discard changes to data dictionary
filepath       Full path and file name of data dictionary
getSection     Return Simulink.data.dictionary.Section object to
               represent data dictionary section
hide           Remove data dictionary from Model Explorer
importEnumTypes Import enumerated type definitions to data dictionary
importFromBaseWorkspace Import base workspace variables to data dictionary
listEntry      List data dictionary entries
removeDataSource Remove reference data dictionary from parent data
dictionary
saveChanges    Save changes to data dictionary
show           Show data dictionary in Model Explorer

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects
(MATLAB).

Examples

Create New Data Dictionary and Data Dictionary Object

Create a data dictionary file myNewDictionary.sldd and a
Simulink.data.Dictionary object representing the new data dictionary. Assign the
object to variable dd1.

dd1 = Simulink.data.dictionary.create('myNewDictionary.sldd')
dd1 =

data dictionary with properties:

    DataSources: {0x1 cell}
    HasUnsavedChanges: 0
    NumberOfEntries: 0

Open Existing Data Dictionary

Create a Simulink.data.Dictionary object representing the existing data dictionary myDictionary_ex_API.sldd. Assign the object to variable dd2.

dd2 = Simulink.data.dictionary.open('myDictionary_ex_API.sldd')

dd2 =

    Dictionary with properties:

        DataSources: {'myRefDictionary_ex_API.sldd'}
        HasUnsavedChanges: 0
        NumberOfEntries: 4

See Also

Simulink.data.dictionary.Entry | Simulink.data.dictionary.Section | Simulink.data.dictionary.create | Simulink.data.dictionary.dictionary.open

Topics

"Store Data in Dictionary Programmatically"
"What Is a Data Dictionary?"

Introduced in R2015a
addDataSource

**Class:** Simulink.data.Dictionary  
**Package:** Simulink.data

Add reference data dictionary to parent data dictionary

**Syntax**

```
addDataSource(dictionaryObj, refDictionaryFile)
```

**Description**


The parent dictionary contains all the entries that are defined in the referenced dictionary until the referenced dictionary is removed from the parent dictionary. The **DataSource** property of an entry indicates the dictionary that defines the entry.

**Input Arguments**

- **dictionaryObj — Parent data dictionary**  
  Simulink.data.Dictionary object

  Parent data dictionary, specified as a Simulink.data.Dictionary object. Before you use this function, represent the target dictionary with a Simulink.data.Dictionary object by using, for example, the **Simulink.data.dictionary.create** or **Simulink.data.dictionary.open** function.

- **refDictionaryFile — File name of data dictionary to reference**  
  character vector

  File name of data dictionary to reference, specified as a character vector that includes the .sldd extension. The data dictionary file must be on your MATLAB path.
Example: 'mySubDictionary_ex_API.sldd'
Data Types: char

Examples

Add a Reference Data Dictionary to a Parent Data Dictionary

Open the data dictionary \texttt{myDictionary\_ex\_API.sldd} and represent it with a \texttt{Simulink.data.Dictionary} object named \texttt{myDictionaryObj}.

\begin{verbatim}
myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');
\end{verbatim}

Add the data dictionary \texttt{mySubDictionary\_ex\_API.sldd} as a reference dictionary to \texttt{myDictionary\_ex\_API.sldd}.

\begin{verbatim}
addDataSource(myDictionaryObj,'mySubDictionary_ex_API.sldd');
\end{verbatim}

Confirm the addition by viewing the \texttt{DataSources} property of variable \texttt{myDictionaryObj}. The property returns the name of the newly referenced dictionary.

\begin{verbatim}
myDictionaryObj.DataSources
\end{verbatim}

\begin{verbatim}
ans =
    'myRefDictionary_ex_API.sldd'
    'mySubDictionary_ex_API.sldd'
\end{verbatim}

Alternatives

You can use the Model Explorer window to manage reference dictionaries. See “Partition Dictionary Data Using Referenced Dictionaries” for more information.

See Also

\texttt{Simulink.data.Dictionary} | \texttt{removeDataSource}

Topics

“Store Data in Dictionary Programmatically”
Introduced in R2015a
close

Class: Simulink.data.Dictionary
Package: Simulink.data

Close connection between data dictionary and Simulink.data.Dictionary object

Syntax

close(dictionaryObj)

Description

close(dictionaryObj) closes the connection between the Simulink.data.Dictionary object dictionaryObj and the data dictionary it represents. dictionaryObj remains as a Simulink.data.Dictionary object but no longer represents any data dictionary.

Input Arguments

dictionaryObj — Target Simulink.data.Dictionary object
handle to Simulink.data.Dictionary object

Target Simulink.data.Dictionary object, specified as a handle to the object.

Tips

- Use the close function in a custom MATLAB function to disassociate a Simulink.data.Dictionary object from a data dictionary. Custom MATLAB functions can create and store variables and objects in function workspaces but cannot delete those variables and objects.
- The close function does not affect the content or the state of the represented data dictionary. The function does not discard unsaved changes to the represented dictionary or entries. You can save or discard them later.
See Also
Simulink.data.Dictionary

Topics
“Store Data in Dictionary Programmatically”

Introduced in R2015a
discardChanges

**Class:** Simulink.data.Dictionary  
**Package:** Simulink.data

Discard changes to data dictionary

**Syntax**

discardChanges(dictionaryObj)

**Description**

discardChanges(dictionaryObj) discards all changes made to the specified data dictionary since the last time changes to the dictionary were saved using the saveChanges function. discardChanges also discards changes made to referenced data dictionaries. The changes to the target dictionary and its referenced dictionaries are permanently lost.

**Input Arguments**

dictionaryObj — Target data dictionary  
Simulink.data.Dictionary object

Target data dictionary, specified as a Simulink.data.Dictionary object. Before you use this function, represent the target dictionary with a Simulink.data.Dictionary object by using, for example, the Simulink.data.dictionary.create or Simulink.data.dictionary.open function.

**Examples**
**Discard Changes to Data Dictionary**

Create a Simulink.data.Dictionary object representing the data dictionary `myDictionary_ex_API.sldd` and assign the object to variable `myDictionaryObj`.

```matlab
myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd')
myDictionaryObj =
  Dictionary with properties:
    DataSources: {'myRefDictionary_ex_API.sldd'}
    HasUnsavedChanges: 0
    NumberOfEntries: 4
```

Make a change to `myDictionary_ex_API.sldd` by adding an entry named `myNewEntry` with value 237. View the HasUnsavedChanges property of `myDictionaryObj` to confirm a change was made.

```matlab
addEntry(dDataSectObj,'myNewEntry',237);
myDictionaryObj

myDictionaryObj =
  Dictionary with properties:
    DataSources: {'myRefDictionary_ex_API.sldd'}
    HasUnsavedChanges: 1
    NumberOfEntries: 5
```

Discard all changes to `myDictionary_ex_API.sldd`. The HasUnsavedChanges property of `myDictionaryObj` indicates changes were discarded.

```matlab
discardChanges(myDictionaryObj)
myDictionaryObj

myDictionaryObj =
  Dictionary with properties:
    DataSources: {'myRefDictionary_ex_API.sldd'}
```
Alternatives

You can use the Model Explorer window to discard changes to data dictionaries. See “View and Revert Changes to Dictionary Entries” for more information.

See Also

Simulink.data.Dictionary | saveChanges

Topics

“Store Data in Dictionary Programmatically”
“What Is a Data Dictionary?”

Introduced in R2015a
filepath

Class: Simulink.data.Dictionary
Package: Simulink.data

Full path and file name of data dictionary

Syntax

dictionaryFilePath = filepath(dictionaryObj)

Description

dictionaryFilePath = filepath(dictionaryObj) returns the full path and file name of the data dictionary dictionaryObj, a Simulink.data.Dictionary object.

Input Arguments

dictionaryObj — Target data dictionary
Simulink.data.Dictionary object

Target data dictionary, specified as a Simulink.data.Dictionary object. Before you use this function, represent the target dictionary with a Simulink.data.Dictionary object by using, for example, the Simulink.data.dictionary.create or Simulink.data.dictionary.open function.

Examples

Return Path of Data Dictionary File

Open the data dictionary myDictionary_ex_API.sldd and represent it with a Simulink.data.Dictionary object named myDictionaryObj.

myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');
Return the full path of `myDictionary_ex_API.sldd` and assign it to variable `myDictionaryFilePath`.

```markdown
myDictionaryFilePath = filepath(myDictionaryObj)
```

```markdown
myDictionaryFilePath =
```

```markdown
C:\Users\jsmith\myDictionary_ex_API.sldd
```

**See Also**
Simulink.data.Dictionary

**Topics**
“Store Data in Dictionary Programmatically”

**Introduced in R2015a**
getSection

Class: Simulink.data.Dictionary
Package: Simulink.data

Return Simulink.data.dictionary.Section object to represent data dictionary section

Syntax

sectionObj = getSection(dictionaryObj,sectionName)

Description

sectionObj = getSection(dictionaryObj,sectionName) returns a Simulink.data.dictionary.Section object representing one section, sectionName, of a data dictionary dictionaryObj, a Simulink.data.Dictionary object.

When you access the Embedded Coder section of a data dictionary, getSection returns a coder.Dictionary object representing the Embedded Coder Dictionary in the data dictionary.

Input Arguments

dictionaryObj — Data dictionary containing target section
Simulink.data.Dictionary object

Data dictionary containing target section, specified as a Simulink.data.Dictionary object. Before you use this function, represent the dictionary with a Simulink.data.Dictionary object by using, for example, the Simulink.data.dictionary.create or Simulink.data.dictionary.open function.

sectionName — Name of target data dictionary section
character vector
Name of target data dictionary section, specified as a character vector.
Example: 'Design Data'
Example: 'Configurations'
Example: 'EmbeddedCoder'
Data Types: char

Examples

Create New Data Dictionary Section Object

Open the data dictionary `myDictionary_ex_API.sldd` and represent it with a `Simulink.data.Dictionary` object named `myDictionaryObj`.

```matlab
myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');
```

Represent the Design Data section of the data dictionary `myDictionary_ex_API.sldd` with a `Simulink.data.dictionary.Section` object named `dDataSectObj`.

```matlab
dDataSectObj = getSection(myDictionaryObj,'Design Data')
```

```
Section with properties:
    Name: 'Design Data'
```

See Also

`Simulink.data.Dictionary` | `Simulink.data.dictionary.Section`

Topics

“Store Data in Dictionary Programmatically”

Introduced in R2015a
hide

Class: Simulink.data.Dictionary
Package: Simulink.data

Remove data dictionary from Model Explorer

Syntax

hide(dictionaryObj)

Description

hide(dictionaryObj) removes the data dictionary dictionaryObj from the Model Hierarchy pane of Model Explorer. The target dictionary no longer appears as a node in the model hierarchy tree. Use this function when you are finished working with a data dictionary and want to reduce clutter in the Model Explorer.

Input Arguments

dictionaryObj — Target data dictionary
Simulink.data.Dictionary object

Target data dictionary, specified as a Simulink.data.Dictionary object. Before you use this function, represent the target dictionary with a Simulink.data.Dictionary object by using, for example, the Simulink.data.dictionary.create or Simulink.data.dictionary.open function.

Examples
**Hide Data Dictionary from Model Explorer**

Open the data dictionary `myDictionary_ex_API.sldd` and represent it with a `Simulink.data.Dictionary` object named `myDictionaryObj`.

```matlab
myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');
```

Open Model Explorer and display the new data dictionary as the selected tree node in the **Model Hierarchy** pane.

```matlab
show(myDictionaryObj)
```

With Model Explorer open, at the MATLAB command prompt, call the `hide` function to observe the removal of `myDictionary_ex_API.sldd` from the model hierarchy tree.

```matlab
hide(myDictionaryObj)
```

**Tips**

- To add a data dictionary as a node in the model hierarchy tree in Model Explorer, use the `show` function or use the interface to open and view the dictionary in Model Explorer.
- The `hide` function does not affect the content of the target dictionary.

**Alternatives**

You can remove a data dictionary from the **Model Hierarchy** pane of Model Explorer by right-clicking the dictionary tree node and selecting **Close**.

**See Also**

`Simulink.data.Dictionary` | `show`

**Topics**

“Store Data in Dictionary Programmatically”

**Introduced in R2015a**
importEnumTypes

Class: Simulink.data.Dictionary
Package: Simulink.data

Import enumerated type definitions to data dictionary

Syntax

importedTypes = importEnumTypes(dictionaryObj, targetTypes)
[importedTypes,importFailures] = importEnumTypes(dictionaryObj, targetTypes)

Description

importedTypes = importEnumTypes(dictionaryObj, targetTypes) imports to the data dictionary dictionaryObj the definitions of one or more enumerated types targetTypes. importEnumTypes does not import MATLAB variables created using enumerated types but instead, in support of those variables, imports the definitions of the types. The target data dictionary stores the definition of a successfully imported type as an entry. This syntax returns a list of the names of successfully imported types. importEnumTypes saves changes made to the target dictionary, so before you use importEnumTypes, confirm that unsaved changes are acceptable.

[importedTypes,importFailures] = importEnumTypes(dictionaryObj, targetTypes) additionally returns a list of any target types that were not successfully imported. You can inspect the list to determine the reason for each failure.

Input Arguments

dictionaryObj — Target data dictionary
Simulink.data.Dictionary object

Target data dictionary, specified as a Simulink.data.Dictionary object. Before you use this function, represent the target dictionary with a Simulink.data.Dictionary object.
object by using, for example, the Simulink.data.dictionary.create or Simulink.data.dictionary.open function.

**targetTypes — Enumerated type definitions to import**
cell array of character vectors | string array

Enumerated type definitions to import, specified as a cell array of character vectors or a string array. If any target types are defined using classdef blocks in MATLAB files or P-files, the files must be available on your MATLAB path so that importEnumTypes can disable them.

Example: {'myEnumType'}
Example: {'myFirstEnumType','mySecondEnumType','myThirdEnumType'}
Data Types: cell

**Output Arguments**

**importedTypes — Target types successfully imported**
array of structures

Target enumerated type definitions successfully imported, returned as an array of structures. Each structure in the array represents one imported type. The className field of each structure identifies a type by name and the renamedFiles field identifies any renamed MATLAB files or P-files.

**importFailures — Target types not imported**
array of structures

Enumerated type definitions targeted but not imported, returned as an array of structures. Each structure in the array represents one type not imported. The className field of each structure identifies a type by name and the reason field explains the failure.

**Examples**
**Import Enumerated Data to Data Dictionary**

Create a data dictionary `myNewDictionary.sldd` in your current working folder and a `Simulink.data.Dictionary` object representing the new data dictionary. Assign the object to the variable `myDictionaryObj`.

```matlab
myDictionaryObj = Simulink.data.dictionary.create('myNewDictionary.sldd');
```

Run the script in the MATLAB file `myDataEnum_ex_API.m`. The file defines an enumerated type named `InstrumentTypes` using the `Simulink.defineIntEnumType` function and creates three variables based on the new type. Then, import the new variables from the base workspace to `myDictionary_ex_API.sldd`.

```matlab
myDataEnum_ex_API
importFromBaseWorkspace(myDictionaryObj,'varList',...
{'firstEnumVariable','secondEnumVariable','thirdEnumVariable'});
```

Clear the imported variables from the base workspace. Before you can import an enumerated data type definition to the target data dictionary, you must clear the base workspace of any variables created using the target type.

```matlab
clear firstEnumVariable
clear secondEnumVariable
clear thirdEnumVariable
```

Import the data type definition to `myDictionary_ex_API.sldd`.

```matlab
importEnumTypes(myDictionaryObj,{’InstrumentTypes’})
```

```matlab
ans =
    className: 'InstrumentTypes'
    renamedFiles: {}
```

**Tips**

- Before you can import an enumerated data type definition to a data dictionary, you must clear the base workspace of any variables created using the target type.
- You can define an enumerated type using a `classdef` block in a MATLAB file or a P-file. `importEnumTypes` imports type definitions directly from these files if you specify the names of the types to import using the input argument `targetTypes` and if the files defining the types are on your MATLAB path.
• To avoid conflicting definitions for imported types, `importEnumTypes` renders MATLAB files or P-files ineffective by appending `.save` to their names. The `.save` extensions cause variables to rely on the definitions in the target data dictionary and not on the definitions in the files. You can remove the `.save` extensions to restore the files to their original state.

• You can use `importEnumTypes` to import enumerated types defined using the `Simulink.defineIntEnumType` function. Because such types are not defined using MATLAB files or P-files, `importEnumTypes` does not rename any files.

• Use the function `Simulink.findVars` to generate a list of the enumerated types that are used by a model. Then, use the list with `importEnumTypes` to import the definitions of the types to a data dictionary. See “Enumerations in Data Dictionary” for more information.

See Also

`Simulink.data.Dictionary` | `importFromBaseWorkspace`

Topics

“Enumerations in Data Dictionary”
“Store Data in Dictionary Programmatically”

Introduced in R2015a
importFromBaseWorkspace

Class: Simulink.data.Dictionary
Package: Simulink.data

Import base workspace variables to data dictionary

Syntax

importedVars = importFromBaseWorkspace(dictionaryObj)
importedVars = importFromBaseWorkspace(dictionaryObj,Name,Value)
[importedVars,existingVars] = importFromBaseWorkspace(____)

Description

importedVars = importFromBaseWorkspace(dictionaryObj) imports all variables from the MATLAB base workspace to the data dictionary dictionaryObj without overwriting existing entries in the dictionary. If any base workspace variables are already in the dictionary, the function present a warning and a list.

This syntax returns a list of names of the successfully imported variables. A variable is considered successfully imported only if importFromBaseWorkspace assigns the value of the variable to the corresponding entry in the target data dictionary.

importedVars = importFromBaseWorkspace(dictionaryObj,Name,Value) imports base workspace variables to a data dictionary, with additional options specified by one or more Name,Value pair arguments.

[importedVars,existingVars] = importFromBaseWorkspace(____) additionally returns a list of variables that were not overwritten. Use this syntax if existingVarsAction is set to 'none', the default value, which prevents existing dictionary entries from being overwritten.
Input Arguments

dictionaryObj — Target data dictionary
Simulink.data.Dictionary object

Target data dictionary, specified as a Simulink.data.Dictionary object. Before you use this function, represent the target dictionary with a Simulink.data.Dictionary object by using, for example, the Simulink.data.dictionary.create or Simulink.data.dictionary.open function.

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside quotes. You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

clearWorkspaceVars — Flag to clear base workspace of imported variables
false (default) | true

Flag to clear the base workspace of any successfully imported variables, specified as the comma-separated pair consisting of 'clearWorkspaceVars' and true or false.

Example: 'clearWorkspaceVars',true

Data Types: logical

existingVarsAction — Action to take for existing dictionary variables
'none' (default) | 'error' | 'overwrite'

Action to take for existing dictionary variables, specified as the comma-separated pair consisting of 'existingVarsAction' and 'none', 'error', or 'overwrite'.

If you specify 'none', importFromBaseWorkspace attempts to import target variables but does not import or make any changes to variables that are already in the data dictionary.

If you specify 'error', importFromBaseWorkspace returns an error, without importing any variables, if any target variables are already in the data dictionary.

If you specify 'overwrite', importFromBaseWorkspace imports all target variables and overwrites any variables that are already in the data dictionary.
Example: 'existingVarsAction','error'

Data Types: char

**varList** — Variables to import
cell array of character vectors | string array

Names of specific base workspace variables to import, specified as the comma-separated pair consisting of 'varList' and a cell array of character vectors or a string array. If you want to import only one variable, specify the name inside a cell array. If you do not specify 'varList', `importFromBaseWorkspace` imports all variables from the MATLAB base workspace.

Example: 'varList',{ 'a','myVariable','fuelFlow'}
Example: 'varList',{ 'fuelFlow'}

Data Types: cell

**Output Arguments**

**importedVars** — Successfully imported variables
cell array of character vectors

Names of successfully imported variables, returned as a cell array of character vectors. A variable is considered successfully imported only if `importFromBaseWorkspace` assigns the value of the variable to the corresponding entry in the target data dictionary.

**existingVars** — Variables that were not imported
cell array of character vectors

Names of target variables that were not imported due to their existence in the target data dictionary, returned as a cell array of character vectors. `existingVars` has content only if 'existingVarsAction' is set to 'none' which is also the default. In that case `importFromBaseWorkspace` imports only variables that are not already in the target data dictionary.

**Examples**
Import All Base Workspace Variables to Data Dictionary

In the MATLAB base workspace, create variables to import.

```matlab
a = 'Char Variable';
myVariable = true;
fuelFlow = 324;
```

Open the data dictionary `myDictionary_ex_API.sldd` and represent it with a `Simulink.data.Dictionary` object named `myDictionaryObj`.

```matlab
myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');
```

Import all base workspace variables to the data dictionary and return a list of successfully imported variables. If any base workspace variables are already in `myDictionary_ex_API.sldd`, `importFromBaseWorkspace` presents a warning and a list of the affected variables.

```matlab
importFromBaseWorkspace(myDictionaryObj);
```

Warning: The following variables were not imported because they already exist in the dictionary:

```matlab
fuelFlow
```

Specify Variables to Import to Data Dictionary from Base Workspace

In the MATLAB base workspace, create variables to import.

```matlab
b = 'Char Variable';
mySecondVariable = true;
airFlow = 324;
```

Open the data dictionary `myDictionary_ex_API.sldd` and represent it with a `Simulink.data.Dictionary` object named `myDictionaryObj`.

```matlab
myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');
```

Import only the new base workspace variables to the data dictionary.

```matlab
importFromBaseWorkspace(myDictionaryObj,'varList',... {'b','mySecondVariable','airFlow'});
```
Import Variables from Base Workspace and Overwrite Conflicts

In the MATLAB base workspace, create a variable to import.

```matlab
fuelFlow = 324;
```


```matlab
myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');
```

Import the variable `fuelFlow` and overwrite the corresponding entry in `myDictionary_ex_API.sldd`.

```matlab
importFromBaseWorkspace(myDictionaryObj,'varList',{'fuelFlow'},...
    'existingVarsAction','overwrite');
```

`importFromBaseWorkspace` assigns the value of the base workspace variable `fuelFlow` to the value of the corresponding entry in `myDictionary_ex_API.sldd`.

Return Variables Not Imported to Data Dictionary from Base Workspace

Return a list of variables that are not imported from the MATLAB base workspace because they are already in the target data dictionary.

In the MATLAB base workspace, create variables to import.

```matlab
fuelFlow = 324;  
myNewVariable = 'This is a character vector.'
```


```matlab
myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');
```

Import the variables `fuelFlow` and `myNewVariable` to the data dictionary. Specify names for the output arguments of `importFromBaseWorkspace` to return the names of successfully and unsuccessfully imported variables.

```matlab
[importedVars,existingVars] = importFromBaseWorkspace(myDictionaryObj,...
    'varList',
    {'fuelFlow','myNewVariable'})
```
importedVars =
    'myNewVariable'

existingVars =
    'fuelFlow'

importFromBaseWorkspace does not import the variable fuelflow because it is already in the target data dictionary.

**Tips**

- importFromBaseWorkspace can import MATLAB variables created from enumerated data types but cannot import the definitions of the enumerated types. Use the importEnumTypes function to import enumerated data type definitions to a data dictionary. If you import variables of enumerated data types to a data dictionary but do not import the enumerated type definitions, the dictionary is less portable and might not function properly if used by someone else.

**Alternatives**

- When you use the Simulink Editor to link a model to a data dictionary, you can choose to import model variables from the base workspace. See “Migrate Single Model to Use Dictionary” for more information.
- You can also use the Model Explorer window to drag-and-drop variables from the base workspace into a data dictionary.

**See Also**

Simulink.data.Dictionary | importEnumTypes

**Topics**

“Store Data in Dictionary Programmatically”

**Introduced in R2015a**
listEntry

**Class:** Simulink.data.Dictionary  
**Package:** Simulink.data

List data dictionary entries

**Syntax**

listEntry(dictionaryObj)  
listEntry(dictionaryObj,Name,Value)

**Description**

listEntry(dictionaryObj) displays in the MATLAB Command Window a table of information about all the entries in the data dictionary dictionaryObj, a Simulink.data.Dictionary object. The displayed information includes the name of each entry, the name of the section containing each entry, the status of each entry, the date and time each entry was last modified, the last user name to modify each entry, and the class of the value each entry contains. By default, the function sorts the list of entries alphabetically by entry name.

listEntry(dictionaryObj,Name,Value) displays the entries in a data dictionary with additional options specified by one or more Name,Value pair arguments.

To return the value of a data dictionary entry at the command prompt, use the getValue method of a Simulink.data.dictionary.Entry object. See “Store Data in Dictionary Programmatically”.

**Input Arguments**

dictionaryObj — Target data dictionary  
Simulink.data.Dictionary object  

Target data dictionary, specified as a Simulink.data.Dictionary object. Before you use this function, represent the target dictionary with a Simulink.data.Dictionary
object by using, for example, the `Simulink.data.dictionary.create` or `Simulink.data.dictionary.open` function.

**Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

**Ascending — Sort order of list**

```
true (default) | false
```

Sort order of the list of data dictionary entries, specified as the comma-separated pair consisting of 'Ascending' and `true` or `false`. If you specify `false`, `listEntry` sorts the list in descending order.

Example: 'Ascending',true

Data Types: logical

**Class — Criteria to filter list by class**

```
character vector
```

Criteria to filter the list of data dictionary entries by class, specified as the comma-separated pair consisting of 'Class' and a character vector identifying a valid class. The function lists only entries whose values are of the specified class.

Example: 'Class','Simulink.Parameter'

Data Types: char

**LastModifiedBy — Criteria to filter list by user name of last modifier**

```
character vector
```

Criteria to filter the list of data dictionary entries by the user name of the last user to modify each entry, specified as the comma-separated pair consisting of 'LastModifiedBy' and a character vector identifying the specified user name. The function lists only entries that were last modified by the specified user name.

Example: 'LastModifiedBy','jsmith'

Data Types: char
**Limit — Maximum number of entries to list**
integer

Maximum number of entries to list, specified as the comma-separated pair consisting of 'Limit' and an integer. The function lists up to the specified number of entries starting from the top of the sorted and filtered list.

Example: 'Limit', 9
Data Types: double

**Name — Criteria to filter list by entry name**
character vector

Criteria to filter the list of data dictionary entries by entry name, specified as the comma-separated pair consisting of 'Name' and a character vector defining the filter criteria. You can use an asterisk character, *, as a wildcard to represent any number of characters. The function lists only entries whose names match the filter criteria.

Example: 'Name', 'fuelFlow'
Example: 'Name', 'fuel*'
Data Types: char

**Section — Criteria to filter list by data dictionary section**
character vector

Criteria to filter the list of data dictionary entries by section, specified as the comma-separated pair consisting of 'Section' and a character vector identifying the target section. The function lists only entries that are contained in the target section.

Example: 'Section', 'Design Data'

**SortBy — Flag to sort list by specific property**

'Name' (default) | 'Section' | 'LastModified' | 'LastModifiedBy'

Flag to sort the list of data dictionary entries by a specific property, specified as the comma-separated pair consisting of 'SortBy' and a character vector identifying a property in the list of entries. Valid properties include 'Name', 'Section', 'LastModified', and 'LastModifiedBy'.

Example: 'SortBy', 'LastModifiedBy'
Examples

List All Entries in Data Dictionary

Represent the data dictionary sldemo_fuelsys_dd_controller.sldd with a Simulink.data.Dictionary object named myDictionaryObj.

```matlab
myDictionaryObj = Simulink.data.dictionary.open('sldemo_fuelsys_dd_controller.sldd');
```

List all the entries in the data dictionary.

```matlab
listEntry(myDictionaryObj)
```

Sort List of Data Dictionary Entries in Descending Order

Represent the data dictionary sldemo_fuelsys_dd_controller.sldd with a Simulink.data.Dictionary object named myDictionaryObj.

```matlab
myDictionaryObj = Simulink.data.dictionary.open('sldemo_fuelsys_dd_controller.sldd');
```

List all the entries in the data dictionary and sort the list in descending order by entry name.

```matlab
listEntry(myDictionaryObj,'Ascending',false)
```

Filter List of Data Dictionary Entries by Name

Represent the data dictionary sldemo_fuelsys_dd_controller.sldd with a Simulink.data.Dictionary object named myDictionaryObj.

```matlab
myDictionaryObj = Simulink.data.dictionary.open('sldemo_fuelsys_dd_controller.sldd');
```

List only the entries in the data dictionary whose names begin with max.

```matlab
listEntry(myDictionaryObj,'Name','max*')
```
Sort List of Data Dictionary Entries by Time of Modification

Represent the data dictionary sldemo_fuelsys_dd_controller.sldd with a Simulink.data.Dictionary object named myDictionaryObj.

myDictionaryObj = Simulink.data.dictionary.open('sldemo_fuelsys_dd_controller.sldd');

List all the entries in the dictionary and sort the list by the date and time each entry was last modified.

listEntry(myDictionaryObj,'SortBy','LastModified')

See Also
Simulink.data.Dictionary | Simulink.data.dictionary.Entry | evalin

Topics
“Store Data in Dictionary Programmatically”
“What Is a Data Dictionary?”

Introduced in R2015a
removeDataSource

Class: Simulink.data.Dictionary
Package: Simulink.data

Remove reference data dictionary from parent data dictionary

Syntax

removeDataSource(dictionaryObj,refDictionaryFile)

Description


The parent dictionary no longer contains the entries that are defined in the referenced dictionary.

Input Arguments

dictionaryObj — Parent data dictionary
Simulink.data.Dictionary object

Parent data dictionary, specified as a Simulink.data.Dictionary object. Before you use this function, represent the target dictionary with a Simulink.data.Dictionary object by using, for example, the Simulink.data.dictionary.create or Simulink.data.dictionary.open function.

refDictionaryFile — File name of referenced data dictionary
character vector

File name of referenced data dictionary, specified as a character vector that includes the .slldd extension. The data dictionary file must be on your MATLAB path.
Example: 'myRefDictionary_ex_API.sldd'
Data Types: char

Examples

Remove Referenced Data Dictionary from Parent Data Dictionary

Open the data dictionary myDictionary_ex_API.sldd and represent it with a Simulink.data.Dictionary object named myDictionaryObj. The DataSources property of myDictionaryObj indicates myDictionary_ex_API.sldd references myRefDictionary_ex_API.sldd.

myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd')

myDictionaryObj =

    Dictionary with properties:
    DataSources: {'myRefDictionary_ex_API.sldd'}
    HasUnsavedChanges: 0
    NumberOfEntries: 4

Remove myRefDictionary_ex_API.sldd from myDictionary_ex_API.sldd.

removeDataSource(myDictionaryObj,'myRefDictionary_ex_API.sldd');

View the properties of the Simulink.data.Dictionary object myDictionaryObj, which represents the parent data dictionary. The DataSources property confirms the removal of myRefDictionary_ex_API.sldd.

myDictionaryObj

myDictionaryObj =

    Dictionary with properties:
    DataSources: {0x1 cell}
Alternatives

You can use Model Explorer to manage reference dictionaries. See “Partition Dictionary Data Using Referenced Dictionaries” for more information.

See Also

Simulink.data.Dictionary | addDataSource

Topics

“Store Data in Dictionary Programmatically”

Introduced in R2015a
saveChanges

Class: Simulink.data.Dictionary
Package: Simulink.data

Save changes to data dictionary

Syntax

saveChanges(dictionaryObj)

Description

saveChanges(dictionaryObj) saves all changes made to a data dictionary dictionaryObj, a Simulink.data.Dictionary object. saveChanges also saves changes made to referenced data dictionaries. The previous states of the target dictionary and its referenced dictionaries are permanently lost.

Input Arguments

dictionaryObj — Target data dictionary
Simulink.data.Dictionary object

Target data dictionary, specified as a Simulink.data.Dictionary object. Before you use this function, represent the target dictionary with a Simulink.data.Dictionary object by using, for example, the Simulink.data.dictionary.create or Simulink.data.dictionary.open function.

Examples
Save Changes to Data Dictionary

Create a new data dictionary myNewDictionary.sldd and represent the Design Data section with a Simulink.data.dictionary.Section object named dDataSectObj.

```matlab
myDictionaryObj = Simulink.data.dictionary.create('myNewDictionary.sldd')
dDataSectObj = getSection(myDictionaryObj,'Design Data');
```

myDictionaryObj =

data dictionary with properties:

- DataSources: {0x1 cell}
- HasUnsavedChanges: 0
- NumberOfEntries: 0

Change myNewDictionary.sldd by adding an entry named myNewEntry with value 237. View the HasUnsavedChanges property of myDictionaryObj to confirm a change was made.

```matlab
addEntry(dDataSectObj,'myNewEntry',237);
```

myDictionaryObj

myDictionaryObj =

Dictionary with properties:

- DataSources: {0x1 cell}
- HasUnsavedChanges: 1
- NumberOfEntries: 1

Save all changes to myNewDictionary.sldd. The HasUnsavedChanges property of myDictionaryObj indicates changes were saved.

```matlab
saveChanges(myDictionaryObj)
```

myDictionaryObj

myDictionaryObj =

Dictionary with properties:

- DataSources: {0x1 cell}
Alternatives

You can use Model Explorer to save changes to a data dictionary by right-clicking on the dictionary tree node in the Model Hierarchy pane and selecting Save Changes.

See Also
Simulink.data.Dictionary | discardChanges

Topics
“Store Data in Dictionary Programmatically”

Introduced in R2015a
**show**

**Class:** Simulink.data.Dictionary  
**Package:** Simulink.data

Show data dictionary in Model Explorer

**Syntax**

show(dictionaryObj)  
show(dictionaryObj,openModelExplorer)

**Description**

`show(dictionaryObj)` opens Model Explorer and displays the data dictionary `dictionaryObj` as the selected tree node in the Model Hierarchy pane.

`show(dictionaryObj,openModelExplorer)` enables you to add the target dictionary to the Model Hierarchy pane without opening Model Explorer.

**Input Arguments**

`dictionaryObj — Target data dictionary`
Simulink.data.Dictionary object

Target data dictionary, specified as a Simulink.data.Dictionary object. Before you use this function, represent the target dictionary with a Simulink.data.Dictionary object by using, for example, the Simulink.data.dictionary.create or Simulink.data.dictionary.open function.

`openModelExplorer — Flag to open Model Explorer`
true (default) | false

Flag to open Model Explorer, specified as true or false.

Data Types: logical
Examples

Show Data Dictionary in Model Explorer

Open the data dictionary myDictionary_ex_API.sldd and represent it with a Simulink.data.Dictionary object named myDictionaryObj.

myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');

Open Model Explorer and display myDictionary_ex_API as the selected node of the model hierarchy tree in the Model Hierarchy pane.

show(myDictionaryObj)

Add Data Dictionary to Model Hierarchy Tree

Open the data dictionary myDictionary_ex_API.sldd and represent it with a Simulink.data.Dictionary object named myDictionaryObj.

myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');

Add myDictionary_ex_API.sldd to the model hierarchy tree without opening Model Explorer.

show(myDictionaryObj,false)

You can confirm the addition of myDictionary_ex_API to the model hierarchy tree by manually opening Model Explorer.

Tips

- Use the hide function to remove a data dictionary from the tree in the Model Hierarchy pane of Model Explorer. The dictionary does not appear in the hierarchy again until you use the show function or you open and view the dictionary in the Model Explorer using the interface.
See Also
Simulink.data.Dictionary

Topics
“Store Data in Dictionary Programmatically”

Introduced in R2015a
Simulink.data.dictionary.Entry class

Package: Simulink.data.dictionary

Configure data dictionary entry

Description

An object of the Simulink.data.dictionary.Entry class represents one entry of a data dictionary. The object allows you to perform operations such as assign the entry a value or change the name of the entry.

Before you can create a new Simulink.data.dictionary.Entry object, you must create a Simulink.data.dictionary.Section object representing the data dictionary section that contains the target entry. However, once created, the Simulink.data.dictionary.Entry object exists independently of the Simulink.data.dictionary.Section object. Use the function getSection to create a Simulink.data.dictionary.Section object.

Construction

The functions addEntry, getEntry, and find create Simulink.data.dictionary.Entry objects.

Properties

DataSource — File name of containing data dictionary
character vector

File name of containing data dictionary, specified as a character vector. Changes you make to this property affect the represented data dictionary entry.

Example: 'myDictionary.sldd'

Data Types: char
**LastModified — Date and time of last modification**
character vector

Date and time of last modification to entry, returned in Coordinated Universal Time (UTC) as a character vector. This property is read only.

**LastModifiedBy — Name of last user to modify entry**
character vector

Name of last user to modify entry, returned as a character vector. This property is read only.

**Name — Name of entry**
character vector

Name of entry, specified as a character vector. Changes you make to this property affect the represented data dictionary entry.

Data Types: char

**Status — State of entry**
'New' | 'Modified' | 'Unchanged' | 'Deleted'

State of entry, returned as 'New', 'Modified', 'Unchanged', or 'Deleted'. The state is valid since the last data dictionary save. If the state is 'Deleted', the represented entry was deleted from its data dictionary. This property is read only.

**Methods**

- `deleteEntry` Delete data dictionary entry
- `discardChanges` Discard changes to data dictionary entry
- `find` Search in array of data dictionary entries
- `getValue` Return value of data dictionary entry
- `setValue` Set value of data dictionary entry
- `showChanges` Display changes made to data dictionary entry
Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects (MATLAB).

Examples

Add Entry to Data Dictionary and Modify its Value

Represent the Design Data section of the data dictionary myDictionary_ex_API.sldd with a Simulink.data.dictionary.Section object named dDataSectObj.

myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');
dDataSectObj = getSection(myDictionaryObj,'Design Data');

Add an entry myEntry with value 27 to the Design Data section of myDictionary_ex_API.sldd. Assign the returned Simulink.data.dictionary.Entry object to variable e.

e = addEntry(dDataSectObj,'myEntry',27)

   e =

    Entry with properties:

    Name: 'myEntry'
    Value: 27
    DataSource: 'myDictionary_ex_API.sldd'
    LastModified: '2014-Aug-26 18:42:08.439709'
    LastModifiedBy: 'jsmith'
    Status: 'New'

Change the value of myEntry from 27 to the character vector 'My New Value'.

setValue(e,'My New Value')

e

   e =

    Entry with properties:
Return Value of Data Dictionary Entry

Represent the data dictionary entry fuelFlow with a Simulink.data.dictionary.Entry object named fuelFlowObj. fuelFlow is defined in the data dictionary myDictionary_ex_API.sldd.

myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');
dDataSectObj = getSection(myDictionaryObj,'Design Data');
fuelFlowObj = getEntry(dDataSectObj,'fuelFlow');

Return the value of the entry fuelFlow and assign the value to the variable fuelFlowValue.

fuelFlowValue = getValue(fuelFlowObj)
fuelFlowValue =

237

Move Entry Within Data Dictionary Hierarchy

Represent the Design Data section of the data dictionary myDictionary_ex_API.sldd with a Simulink.data.dictionary.Section object named dDataSectObj. myDictionary_ex_API.sldd references the data dictionary myRefDictionary_ex_API.sldd.

myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');
dDataSectObj = getSection(myDictionaryObj,'Design Data');

Create a Simulink.data.dictionary.Entry object representing the entry fuelFlow, which resides in myDictionary_ex_API.sldd. Assign the object to variable e.

e = getEntry(dDataSectObj,'fuelFlow')
e =

Entry with properties:

   Name: 'fuelFlow'
   Value: 237
   DataSource: 'myDictionary_ex_API.sldd'
   LastModified: '2014-Sep-05 13:12:06.099278'
   LastModifiedBy: 'jsmith'
   Status: 'Unchanged'

Migrate the entry fuelFlow to the reference data dictionary myRefDictionary_ex_API.sldd by modifying the DataSource property of e.

e.DataSource = 'myRefDictionary_ex_API.sldd'

e =

Entry with properties:

   Name: 'fuelFlow'
   Value: 237
   DataSource: 'myRefDictionary_ex_API.sldd'
   LastModified: '2014-Sep-05 13:12:06.099278'
   LastModifiedBy: 'jsmith'
   Status: 'Modified'

Because myDictionary_ex_API.sldd references myRefDictionary_ex_API.sldd, both dictionaries belong to the same dictionary hierarchy, allowing you to migrate the entry fuelFlow between them.

See Also
Simulink.data.Dictionary|Simulink.data.dictionary.Section|getEntry

Topics
“Store Data in Dictionary Programmatically”
“What Is a Data Dictionary?”

Introduced in R2015a
deleteEntry

Class: Simulink.data.dictionary.Entry
Package: Simulink.data.dictionary

Delete data dictionary entry

Syntax

deleteEntry(entryObj)

Description

deleteEntry(entryObj) deletes the data dictionary entry represented by entryObj, a Simulink.data.dictionary.Entry object. The represented entry no longer exists in the data dictionary that defined it.

The function sets the Status properties of any Simulink.data.dictionary.Entry objects representing the deleted entry to 'Deleted'. You can access only the Status properties of the objects.

Input Arguments

entryObj — Target data dictionary entry
Simulink.data.dictionary.Entry object

Target data dictionary entry, specified as a Simulink.data.dictionary.Entry object. Before you use this function, represent the target entry with a Simulink.data.dictionary.Entry object by using, for example, the getEntry function.

Examples
**Delete Data Dictionary Entry**

Represent the data dictionary entry `fuelFlow` with a `Simulink.data.dictionary.Entry` object named `fuelFlowObj`. `fuelFlow` is defined in the data dictionary `myDictionary_ex_API.sldd`.

```matlab
myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');
dDataSectObj = getSection(myDictionaryObj,'Design Data');
fuelFlowObj = getEntry(dDataSectObj,'fuelFlow');
```

Delete the entry `fuelFlow` from the data dictionary `myDictionary_ex_API.sldd`. `myDictionary_ex_API.sldd` no longer contains the `fuelFlow` entry.

```matlab
deleteEntry(fuelFlowObj)
```

**Alternatives**

You can use the Model Explorer window to view the contents of a data dictionary and delete entries.

**See Also**

`Simulink.data.dictionary.Entry`, `addEntry`

**Topics**

“Store Data in Dictionary Programmatically”

**Introduced in R2015a**
discardChanges

Class: Simulink.data.dictionary.Entry
Package: Simulink.data.dictionary

Discard changes to data dictionary entry

Syntax

discardChanges(entryObj)

Description

discardChanges(entryObj) discards all changes made to the data dictionary entry entryObj, a Simulink.data.dictionary.Entry object, since the last time the containing data dictionary was saved using the saveChanges function. The changes to the entry are permanently lost.

Input Arguments

entryObj — Target data dictionary entry
Simulink.data.dictionary.Entry object

Target data dictionary entry, specified as a Simulink.data.dictionary.Entry object. Before you use this function, represent the target entry with a Simulink.data.dictionary.Entry object by using, for example, the getEntry function.

Examples
Discard Changes to Data Dictionary Entry

Represent the data dictionary entry fuelFlow with a Simulink.data.dictionary.Entry object named fuelFlowObj. fuelFlow is defined in the data dictionary myDictionary_ex_API.sldd.

```
myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');
dDataSectObj = getSection(myDictionaryObj,'Design Data');
fuelFlowObj = getEntry(dDataSectObj,'fuelFlow');
```

Change the entry fuelFlow by assigning it the new value 493. Confirm a change was made by viewing the Status property of fuelFlowObj.

```
setValue(fuelFlowObj,493);
fuelFlowObj
```

Entry with properties:

- Name: 'fuelFlow'
- Value: 493
- DataSource: 'myDictionary_ex_API.sldd'
- LastModified: '2014-Sep-05 13:14:30.661978'
- LastModifiedBy: 'jsmith'
- Status: 'Modified'

Discard all changes to the entry fuelFlow. The Status property of fuelFlowObj shows that changes were discarded.

```
discardChanges(fuelFlowObj)
fuelFlowObj
```

Entry with properties:

- Name: 'fuelFlow'
- Value: 237
- DataSource: 'myDictionary_ex_API.sldd'
- LastModified: '2014-Sep-05 13:12:06.099278'
Tips

- You can use the discardChanges function or the saveChanges function with an entire data dictionary, discarding or saving changes to all entries in the dictionary at once. However, only the discardChanges function can additionally operate on individual entries. You cannot use the saveChanges function to save changes to individual entries.

Alternatives

You can use Model Explorer and the Comparison Tool to discard changes to data dictionary entries. See “View and Revert Changes to Dictionary Entries” for more information.

See Also

Simulink.data.dictionary.Entry | saveChanges

Topics

“Store Data in Dictionary Programmatically”

Introduced in R2015a
find

**Class:** Simulink.data.dictionary.Entry  
**Package:** Simulink.data.dictionary

Search in array of data dictionary entries

**Syntax**

```plaintext
foundEntries = find(targetEntries,PName1,PValue1,...,PNameN,PValueN)
foundEntries = find(targetEntries,PName1,PValue1,...,PNameN,PValueN, options)
```

**Description**

```plaintext
foundEntries = find(targetEntries,PName1,PValue1,...,PNameN,PValueN)
```
sources the array of data dictionary entries `targetEntries` using search criteria `PName1,PValue1,...,PNameN,PValueN`, and returns an array of entries matching the criteria. This syntax matches the search criteria with the properties of the target entries, which are `Simulink.data.dictionary.Entry` objects, but not with the properties of their values. See `Simulink.data.dictionary.Entry` for a list of data dictionary entry properties.

```plaintext
foundEntries = find(targetEntries,PName1,PValue1,...,PNameN,PValueN, options)
```
sources for data dictionary entries using additional search options. For example, you can match the search criteria with the values of the target entries.

**Input Arguments**

- `targetEntries` — **Data dictionary entries to search**
  
  Array of `Simulink.data.dictionary.Entry` objects

  Data dictionary entries to search, specified as an array `Simulink.data.dictionary.Entry` objects. Before you use this function, represent the target entries with `Simulink.data.dictionary.Entry` objects by using, for example, the `getEntry` function.
Example: [myEntryObj1,myEntryObj2,myEntryObj3]

**PName1,PValue1,...,PNameN,PValueN — Search criteria**  
name-value pairs representing properties

Search criteria, specified as one or more name-value pairs representing names and values of properties of the target data dictionary entries. For a list of the properties of a data dictionary entry, see `Simulink.data.dictionary.Entry`. If you specify more than one name-value pair, the returned entries meet all of the criteria.

If you include the '-value' option to search in the values of the target entries, the search criteria apply to the values of the entries rather than to the entries themselves.

Example: 'LastModifiedBy','jsmith'
Example: 'DataSource','myRefDictionary_ex_API.sldd'

**options — Additional search options**  
supported option codes

Additional search options, specified as one or more of the following supported option codes.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'-value'</td>
<td>This option causes find to search only in the values of the target data dictionary entries. Specify this option before any other search criteria or options arguments.</td>
</tr>
<tr>
<td>'-and', '-or', '-xor', or '-not' logical operators</td>
<td>These options modify or combine multiple search criteria or other option codes.</td>
</tr>
<tr>
<td>'-property', propertyName</td>
<td>This name-value pair causes find to search for entries or values that have the property propertyName regardless of the value of the property. Specify propertyName as a character vector.</td>
</tr>
<tr>
<td>'-class', className</td>
<td>This name-value pair causes find to search for entries or values that are objects of the class className. Specify className as a character vector.</td>
</tr>
</tbody>
</table>
'isa',className

This name-value pair causes find to search for entries or values that are objects of the class or of any subclass derived from the class className. Specify className as a character vector.

'-regexp'

This option allows you to use regular expressions in your search criteria. This option affects only search criteria that follow '-regexp'.

Example: '-value'
Example: '-value','-property','CoderInfo'
Example: '-value','-class','Simulink.Parameter'

Output Arguments

foundEntries — Data dictionary entries matching search criteria

array of Simulink.data.dictionary.Entry objects

Data dictionary entries matching the specified search criteria, returned as an array of Simulink.data.dictionary.Entry objects.

Examples

Search Data Dictionary Entry Values for Specific Class

Search in an array of data dictionary entries myEntryObjs for entries whose values are objects of the class Simulink.Parameter.

foundEntries = find(myEntryObjs,'-value','-class','Simulink.Parameter')

Search Data Dictionary Entries for Modifying User

Search in an array of data dictionary entries myEntryObjs for entries that were last modified by the user jsmith.
foundEntries = find(myEntryObjs,'LastModifiedBy','jsmith')

**Search Data Dictionary Entries Using Multiple Criteria**

Search in an array of data dictionary entries myEntryObjs for entries that were last modified by the user jsmith or whose names begin with fuel.

```
foundEntries = find(myEntryObjs,'LastModifiedBy','jsmith','-or','-regexp','Name','fuel*')
```

**Search Data Dictionary Entries Using Regular Expressions**

Search in an array of data dictionary entries myEntryObjs for entries whose names begin with Press.

```
foundEntries = find(myEntryObjs,'-regexp','Name','Press*')
```

**Search Data Dictionary Entries for Specific Value**

Search in an array of data dictionary entries myEntryObjs for entries whose values are 273. If you find more than one entry, store the entries in an array called foundEntries.

```
foundEntries = [];  
for i = 1:length(myEntryObjs)  
    if getValue(myEntryObjs(i)) == 237  
        foundEntries = [foundEntries myEntryObjs(i)];  
    end  
end
```

**Search Data Dictionary Entry Values for Specific Property**

Search in an array of data dictionary entries myEntryObjs for entries whose values have a property DataType.
foundEntries = find(myEntryObjs,'-value','-property','DataType')

See Also
Simulink.data.dictionary.Entry | find

Topics
“Store Data in Dictionary Programmatically”

Introduced in R2015a
GetValue

**Class:** Simulink.data.dictionary.Entry  
**Package:** Simulink.data.dictionary

Return value of data dictionary entry

**Syntax**

entryValue = getValue(entryObj)

**Description**

entryValue = getValue(entryObj) returns the value of the data dictionary entry entryObj, a Simulink.data.dictionary.Entry object.

To programmatically access variables for the purpose of sweeping block parameter values, consider using Simulink.SimulationInput objects instead of modifying the variables through the programmatic interface of the data dictionary. See “Optimize, Estimate, and Sweep Block Parameter Values”.

**Input Arguments**

**entryObj — Target data dictionary entry**
Simulink.data.dictionary.Entry object

Target data dictionary entry, specified as a Simulink.data.dictionary.Entry object. Before you use this function, represent the target entry with a Simulink.data.dictionary.Entry object by using, for example, the getEntry function.

**Examples**
Return Value of Data Dictionary Entry

Represent the data dictionary entry fuelFlow with a Simulink.data.dictionary.Entry object named fuelFlowObj. fuelFlow is defined in the data dictionary myDictionary_ex_API.sldd.

```matlab
myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');
dDataSectObj = getSection(myDictionaryObj,'Design Data');
fuelFlowObj = getEntry(dDataSectObj,'fuelFlow');
```

Return the value of the entry fuelFlow and assign the value to variable fuelFlowValue.

```matlab
fuelFlowValue = getValue(fuelFlowObj)
```

fuelFlowValue = 237

See Also

Simulink.data.dictionary.Entry | setValue

Topics

“Store Data in Dictionary Programmatically”

Introduced in R2015a
**setValue**

**Class:** Simulink.data.dictionary.Entry  
**Package:** Simulink.data.dictionary

Set value of data dictionary entry

**Syntax**

```matlab
definitions setValue(entryObj, newValue)
```  

**Description**

setValue(entryObj, newValue) assigns the value newValue to the data dictionary entry entryObj, a Simulink.data.dictionary.Entry object.

To programmatically access variables for the purpose of sweeping block parameter values, consider using Simulink.SimulationInput objects instead of modifying the variables through the programmatic interface of the data dictionary. See “Optimize, Estimate, and Sweep Block Parameter Values”.

**Input Arguments**

- **entryObj — Target data dictionary entry**  
  Simulink.data.dictionary.Entry object

  Target data dictionary entry, specified as a Simulink.data.dictionary.Entry object. Before you use this function, represent the target entry with a Simulink.data.dictionary.Entry object by using, for example, the getEntry function.

- **newValue — Value to assign to data dictionary entry**  
  MATLAB expression
Value to assign to data dictionary entry, specified as a MATLAB expression. The expression must return a value that is supported by the data dictionary section that contains the entry.

Example: 27.5
Example: myBaseWorkspaceVariable
Example: Simulink.Parameter

Examples

Set Value of Data Dictionary Entry

Represent the data dictionary entry fuelFlow with a Simulink.data.dictionary.Entry object named fuelFlowObj. fuelFlow is defined in the data dictionary myDictionary_ex_API.sldd.

myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');
dDataSectObj = getSection(myDictionaryObj,'Design Data');
fuelFlowObj = getEntry(dDataSectObj,'fuelFlow');

Set the value of the entry fuelFlow to 493. Then, view the Value property of fuelFlowObj to observe the change.

setValue(fuelFlowObj,493)
fuelFlowObj

fuelFlowObj =

Entry with properties:

    Name: 'fuelFlow'
    Value: 493
    DataSource: 'myDictionary_ex_API.sldd'
    LastModified: '2014-Sep-05 13:37:22.161124'
Alternatives

You can use the Model Explorer window to view and change the values of data dictionary entries.

See Also

Simulink.data.dictionary.Entry | getValue

Topics

“Store Data in Dictionary Programmatically”

Introduced in R2015a
showChanges

Class: Simulink.data.dictionary.Entry
Package: Simulink.data.dictionary

Display changes made to data dictionary entry

Syntax

showChanges(entryObj)

Description

showChanges(entryObj) opens the Comparison Tool to show changes made to the data dictionary entry entryObj, a Simulink.data.dictionary.Entry object. The Comparison Tool displays the properties of entryObj as they were when the data dictionary was last saved and as they were when the showChanges function was called.

Input Arguments

entryObj — Target data dictionary entry
Simulink.data.dictionary.Entry object

Target data dictionary entry, specified as a Simulink.data.dictionary.Entry object. Before you use this function, represent the target entry with a Simulink.data.dictionary.Entry object by using, for example, the getEntry function.

Examples
View Unsaved Changes to Data Dictionary Entry

Represent the data dictionary entry `fuelFlow` with a Simulink.data.dictionary.Entry object named `fuelFlowObj`. `fuelFlow` is defined in the data dictionary `myDictionary_ex_API.sldd`.

```matlab
myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');
dDataSectObj = getSection(myDictionaryObj,'Design Data');
fuelFlowObj = getEntry(dDataSectObj,'fuelFlow');
```

Make a change to the entry `fuelFlow` by assigning it the new value 494.

```matlab
setValue(fuelFlowObj,494);
```

Observe the unsaved change to the entry `fuelFlow`. The Comparison Tool opens and compares side by side the current state of the entry with its most recently saved state.

```matlab
showChanges(fuelFlowObj)
```

Alternatives

You can use Model Explorer and the Comparison Tool to view changes to data dictionary entries. See “View and Revert Changes to Dictionary Entries” for more information.

See Also

Simulink.data.dictionary.Entry | discardChanges

Topics

“Store Data in Dictionary Programmatically”

Introduced in R2015a
Simulink.data.dictionary.EnumTypeDefinition class

Package: Simulink.data.dictionary

Store enumerated type definition in data dictionary

Description

An object of the Simulink.data.dictionary.EnumTypeDefinition class defines an enumerated data type in a data dictionary. You store the object in a data dictionary entry so models linked to the dictionary can use the enumerated type definition.

In the MATLAB base workspace, objects of this class retain information about an enumerated type but do not define the type for use by other variables or by models.

Construction

When you use the function `importEnumTypes` to import the definitions of enumerated types to a data dictionary, Simulink creates a Simulink.data.dictionary.EnumTypeDefinition object in the dictionary for each imported definition. The dictionary stores each object in an individual entry.

The constructor Simulink.data.dictionary.EnumTypeDefinition creates an instance of this class with default property values and a single enumeration member that has underlying integer value 0.

Properties

AddClassNameToEnumNames — Flag to control enumeration identifiers in generated code
false (default) | true

Flag to prefix enumerations with the class name in generated code, specified as true or false.
If you specify `true`, when you generate code the identifier of each enumeration member begins with the name of the enumeration class. For example, an enumeration class `LEDcolor` with enumeration members `GREEN` and `RED` defines the enumeration members in generated code as `LEDcolor_GREEN` and `LEDcolor_RED`.

**Data Types:** logical

**DataScope — Flag to control data type definition in generated code**

'Auto' (default) | 'Imported' | 'Exported'

Flag to control data type definition in generated code, specified as 'Auto', 'Imported', or 'Exported'. The table describes the behavior of generated code for each value.

<table>
<thead>
<tr>
<th>Value</th>
<th>Action</th>
</tr>
</thead>
</table>
| Auto (default) | If you do not specify the property `Headerfile`, export the data type definition to `model_types.h`, where `model` is the model name.  
If you specify `Headerfile`, import the data type definition from the specified header file. |
| Exported       | Export the data type definition to a separate header file.             |
|                | If you do not specify the property `Headerfile`, the header file name defaults to `type.h`, where `type` is the data type name. |
| Imported       | Import the data type definition from a separate header file.           |
|                | If you do not specify the property `Headerfile`, the header file name defaults to `type.h`, where `type` is the data type name. |

**DefaultValue — Default enumeration member**

' ' (default) | character vector
Default enumeration member, specified as a character vector. Specify DefaultValue as the name of an enumeration member you have already defined.

When you create a Simulink.data.dictionary.EnumTypeDefinition object, DefaultValue is an empty character vector, '', and Simulink uses the first enumeration member as the default member.

Example: 'enumMember1'

**Description — Description of enumerated data type in generated code**

```
' ' (default) | character vector
```

Description of the enumerated data type, specified as a character vector. Use this property to explain the purpose of the type in generated code.

Example: 'Two possible colors of LED indicator: GREEN and RED.'

Data Types: char

**HeaderFile — Name of header file defining enumerated data type in generated code**

```
' ' (default) | character vector
```

Name of the header file that defines the enumerated data type in generated code, specified as a character vector. Use a .h extension to specify the file name.

If you do not specify HeaderFile, generated code uses a default header file name that depends on the value of the DataScope property.

Example: 'myTypeIncludeFile.h'

Data Types: char

**StorageType — Data type of underlying integer values**

```
' ' (default) | character vector
```

Data type of the integer values underlying the enumeration members, specified as a character vector. Generated code stores the underlying integer values using the data type you specify.

You can specify one of these supported integer types:

- 'int8'
- 'int16'
• 'int32'
• 'uint8'
• 'uint16'

To store the underlying integer values in generated code using the native integer type of the target hardware, specify **StorageType** as an empty character vector, '', which is the default value.

Example: 'int16'


## Methods

appendEnumeral Add enumeration member to enumerated data type definition in data dictionary

removeEnumeral Remove enumeration member from enumerated data type definition in data dictionary

## Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects (MATLAB).

## Examples

### Programmatically Create Enumerated Type Definition in Data Dictionary

Create an object that can store the definition of an enumerated type. By default, the new type defines a single enumeration member `enum1` with underlying integer value 0.

```matlab
myColors = Simulink.data.dictionary.EnumTypeDefinition
myColors =
```
Add some enumeration members to the definition of the type.

```matlab
appendEnumeral(myColors,'Orange',1,'')
appendEnumeral(myColors,'Black',2,'')
appendEnumeral(myColors,'Cyan',3,'')
```

```matlab
myColors
```

```matlab
Simulink.data.dictionary.EnumTypeDefinition
    enum1
    Orange
    Black
    Cyan
```

Remove the default enumeration member `enum1`. Since `enum1` is the first enumeration member in the list, identify it with index 1.

```matlab
removeEnumeral(myColors,1)
```

```matlab
myColors
```

```matlab
Simulink.data.dictionary.EnumTypeDefinition
    Orange
    Black
    Cyan
```

Customize the enumerated type by configuring the properties of the object representing it.

```matlab
myColors.Description = 'These are my favorite colors.';
myColors.DefaultValue = 'Cyan';
myColors.HeaderFile = 'colorsType.h';
```

Open the data dictionary `myDictionary_ex_API.sldd` and represent it with a `Simulink.data.Dictionary` object named `myDictionaryObj`.

```matlab
myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');
```

Import the object that defines the enumerated type `myColors` to the dictionary.
importFromBaseWorkspace(myDictionaryObj,'varList',{('myColors')});

Alternatives

You can use Model Explorer to add and modify enumerated data types stored in a data dictionary.

See Also

Simulink.data.Dictionary

Topics

“Store Data in Dictionary Programmatically”
“Use Enumerated Data in Simulink Models”

Introduced in R2015a
appendEnumeral

Class: Simulink.data.dictionary.EnumTypeDefinition
Package: Simulink.data.dictionary

Add enumeration member to enumerated data type definition in data dictionary

Syntax

appendEnumeral(typeObj,memberName,memberValue,memberDesc)

Description

appendEnumeral(typeObj,memberName,memberValue,memberDesc) adds an enumeration member to the enumerated type definition stored by typeObj, a Simulink.data.dictionary.EnumTypeDefinition object.

Input Arguments

typeObj — Target enumerated type definition
Simulink.data.dictionary.EnumTypeDefinition object

Target enumerated type definition, specified as a Simulink.data.dictionary.EnumTypeDefinition object.

memberName — Name of new enumeration member
character vector

Name of the new enumeration member, specified as a character vector.

Example: 'myNewEnumMember'

Data Types: char

memberValue — Integer value underlying new enumeration member
integer
Integer value underlying the new enumeration member, specified as an integer.

The definition of the enumeration class determines the integer data type used in generated code to store the underlying values of enumeration members.

Example: 3
Data Types: single | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | double

memberDesc — Description of new enumeration member
character vector

Description of the new enumeration member, specified as a character vector.

If you do not want to supply a description for the enumeration member, use an empty character vector.

Example: 'Enumeration member number 1.'
Example: ''
Data Types: char

Examples

Programmatically Create Enumerated Type Definition in Data Dictionary

Create an object that can store the definition of an enumerated type. By default, the new type defines a single enumeration member enum1 with underlying integer value 0.

myColors = Simulink.data.dictionary.EnumTypeDefinition
myColors =
    Simulink.data.dictionary.EnumTypeDefinition
    enum1

Add some enumeration members to the definition of the type.

appendEnumeral(myColors,'Orange',1,'')
appendEnumeral(myColors,'Black',2,'')
appendEnumeral(myColors,'Cyan',3,'')

myColors =

Simulink.data.dictionary.EnumTypeDefinition
    enum1
    Orange
    Black
    Cyan

Remove the default enumeration member enum1. Since enum1 is the first enumeration member in the list, identify it with index 1.

removeEnumeral(myColors,1)

myColors =

Simulink.data.dictionary.EnumTypeDefinition
    Orange
    Black
    Cyan

Customize the enumerated type by configuring the properties of the object representing it.

myColors.Description = 'These are my favorite colors.';
myColors.DefaultValue = 'Cyan';
myColors.HeaderFile = 'colorsType.h';

Open the data dictionary myDictionary_ex_API.sldd and represent it with a Simulink.data.Dictionary object named myDictionaryObj.

myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');

Import the object that defines the enumerated type myColors to the dictionary.
importFromBaseWorkspace(myDictionaryObj,'varList',{myColors});

**Alternatives**

You can use Model Explorer to add enumeration members to the enumerated data type represented by a Simulink.data.dictionary.EnumTypeDefinition object.
See Also
Simulink.data.dictionary.EnumTypeDefinition|
Simulink.data.dictionary.EnumTypeDefinition.removeEnumeral

Topics
“Store Data in Dictionary Programmatically”
“Use Enumerated Data in Simulink Models”

Introduced in R2015a
removeEnumeral

**Class:** Simulink.data.dictionary.EnumTypeDefinition  
**Package:** Simulink.data.dictionary

Remove enumeration member from enumerated data type definition in data dictionary

**Syntax**

```matlab
removeEnumeral(typeObj,memberNum)
```

**Description**

`removeEnumeral(typeObj,memberNum)` removes an enumeration member from the enumerated type definition stored by `typeObj`, a `Simulink.data.dictionary.EnumTypeDefinition` object.

**Input Arguments**

- `typeObj` — Target enumerated type definition  
  `Simulink.data.dictionary.EnumTypeDefinition` object

  Target enumerated type definition, specified as a `Simulink.data.dictionary.EnumTypeDefinition` object.

- `memberNum` — Index of target enumeration member  
  integer

  Index of target enumeration member, specified as an integer.

  The first enumeration member in an enumerated type definition has index 1. For example, suppose an enumerated type `BasicColors` has this definition:

  ```matlab
  myColors =
  Simulink.data.dictionary.EnumTypeDefinition
  ```
Orange
Black
Cyan

To remove the enumeration member Black, specify memberNum as 2. To remove the enumeration member Cyan, specify 3.

Do not specify memberNum using the integer value underlying an enumeration member. The integer value underlying the member is not equivalent to the index of the member.

Example: 3

Data Types: single | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | double

Examples

Programmatically Create Enumerated Type Definition in Data Dictionary

Create an object that can store the definition of an enumerated type. By default, the new type defines a single enumeration member enum1 with underlying integer value 0.

myColors = Simulink.data.dictionary.EnumTypeDefinition

myColors =

    Simulink.data.dictionary.EnumTypeDefinition
    enum1

Add some enumeration members to the definition of the type.

appendEnumeral(myColors,'Orange',1,"
appendEnumeral(myColors,'Black',2,"
appendEnumeral(myColors,'Cyan',3,"

myColors

myColors =

    Simulink.data.dictionary.EnumTypeDefinition
    enum1
    Orange
    Black
    Cyan
Remove the default enumeration member `enum1`. Since `enum1` is the first enumeration member in the list, identify it with index 1.

```plaintext
removeEnumeral(myColors,1)
```

```plaintext
myColors
```

```plaintext
myColors =

    Simulink.data.dictionary.EnumTypeDefinition
    Orange
    Black
    Cyan
```

Customize the enumerated type by configuring the properties of the object representing it.

```plaintext
myColors.Description = 'These are my favorite colors.';
myColors.DefaultValue = 'Cyan';
myColors.HeaderFile = 'colorsType.h';
```

Open the data dictionary `myDictionary_ex_API.sldd` and represent it with a Simulink.data.Dictionary object named `myDictionaryObj`.

```plaintext
myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');
```

Import the object that defines the enumerated type `myColors` to the dictionary.

```plaintext
importFromBaseWorkspace(myDictionaryObj,'varList',{'myColors'});
```

### Alternatives

You can use Model Explorer to remove enumeration members from the enumerated data type represented by a Simulink.data.dictionary.EnumTypeDefinition object.

### See Also

- Simulink.data.dictionary.EnumTypeDefinition
- Simulink.data.dictionary.EnumTypeDefinition.appendEnumeral

### Topics

“Store Data in Dictionary Programmatically”
“Use Enumerated Data in Simulink Models”

*Introduced in R2015a*
Simulink.data.dictionary.Section class

Package: Simulink.data.dictionary

Configure data dictionary section

Description

An object of the Simulink.data.dictionary.Section class represents one section of a data dictionary, such as Design Data or Configurations. The object allows you to perform operations on the section such as add or delete entries and import data from files.

Before you can create a Simulink.data.dictionary.Section object, you must create a Simulink.data.Dictionary object representing the target data dictionary. Once created, the Simulink.data.dictionary.Section object exists independently of the Simulink.data.Dictionary object.

You cannot use the data dictionary programmatic interface (see “Store Data in Dictionary Programmatically”) to access the Embedded Coder section of a data dictionary. Instead, see Embedded Coder Dictionary.

Construction

The function getSection creates a Simulink.data.dictionary.Section object.

Properties

Name — Name of data dictionary section
character vector

Name of data dictionary section, returned as a character vector. This property is read only.
Methods

addEntry      Add new entry to data dictionary section
assignin      Assign value to data dictionary entry
deleteEntry   Delete data dictionary entry
evalin        Evaluate MATLAB expression in data dictionary section
exist         Check existence of data dictionary entry
exportToFile  Export data dictionary entries from section to MAT-file or MATLAB file
find          Search in data dictionary section
getEntry      Create Simulink.data.dictionary.Entry object to represent data dictionary entry
importFromFile Import variables from MAT-file or MATLAB file to data dictionary section

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects (MATLAB).

Examples

Create New Data Dictionary Section Object

Open the data dictionary myDictionary_ex_API.sldd and represent it with a Simulink.data.Dictionary object named myDictionaryObj.

myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');

Represent the Design Data section of the data dictionary myDictionary_ex_API.sldd with a Simulink.data.dictionary.Section object named dDataSectObj.

dDataSectObj = getSection(myDictionaryObj,'Design Data')

dDataSectObj =

  Section with properties:
Name: 'Design Data'

See Also
Simulink.data.Dictionary | getSection

Topics
“Store Data in Dictionary Programmatically”
“What Is a Data Dictionary?”

Introduced in R2015a
addEntry

Class: Simulink.data.dictionary.Section
Package: Simulink.data.dictionary

Add new entry to data dictionary section

Syntax

addEntry(sectionObj,entryName,entryValue)
entryObj = addEntry(sectionObj,entryName,entryValue)

Description

addEntry(sectionObj,entryName,entryValue) adds an entry, with name entryName and value entryValue, to the data dictionary section sectionObj, a Simulink.data.dictionary.Section object.

entryObj = addEntry(sectionObj,entryName,entryValue) returns a Simulink.data.dictionary.Entry object representing the newly added data dictionary entry.

Input Arguments

sectionObj — Target data dictionary section
Simulink.data.dictionary.Section object

Target data dictionary section, specified as a Simulink.data.dictionary.Section object. Before you use this function, represent the target section with a Simulink.data.dictionary.Section object by using, for example, the getSection function.

entryName — Name of new data dictionary entry
character vector

Name of new data dictionary entry, specified as a character vector.
Example: 'myNewEntry'
Data Types: char

**entryValue** — Value of new data dictionary entry
MATLAB expression

Value of new data dictionary entry, specified as a MATLAB expression that returns any valid data dictionary content.
Example: 27.5
Example: myBaseWorkspaceVariable
Example: Simulink.Parameter

## Examples

### Add Entry to Design Data Section of Data Dictionary

Represent the Design Data section of the data dictionary `myDictionary_ex_API.sldd` with a `Simulink.data.dictionary.Section` object named `dDataSectObj`.

```matlab
myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');
dDataSectObj = getSection(myDictionaryObj,'Design Data');
```

Add an entry to the Design Data section of `myDictionary_ex_API.sldd` an entry `myNewEntry` with value 237.

```matlab
addEntry(dDataSectObj,'myNewEntry',237)
```

### Add New Simulink.Parameter Object to Data Dictionary

Represent the Design Data section of the data dictionary `myDictionary_ex_API.sldd` with a `Simulink.data.dictionary.Section` object named `dDataSectObj`.

```matlab
myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');
dDataSectObj = getSection(myDictionaryObj,'Design Data');
```

Add an entry to the Design Data section of `myDictionary_ex_API.sldd`. Name the new entry `myNewParam` and assign a `Simulink.Parameter` object to the value.

```matlab
addEntry(dDataSectObj,'myNewParam', Simulink.Parameter('Value',237))
```
addEntry(dDataSectObj,'myNewParam',Simulink.Parameter)

The expression `Simulink.Parameter` constructs a new `Simulink.Parameter` object, and the `addEntry` function assigns the object to the value of the new data dictionary entry `myNewParam`.

**Tips**

- `addEntry` returns an error if the entry name you specify with `entryName` is already the name of an entry in the target data dictionary section or in the same section of any referenced dictionaries.

**Alternatives**

You can use Model Explorer to add entries to a data dictionary in the same way you can use it to add variables to a model workspace or the base workspace.

**See Also**

- `Simulink.data.dictionary.Entry`
- `Simulink.data.dictionary.Section`
- `assignin`

**Topics**

- “Store Data in Dictionary Programmatically”

**Introduced in R2015a**
assignin

**Class:** Simulink.data.dictionary.Section  
**Package:** Simulink.data.dictionary

Assign value to data dictionary entry

**Syntax**

`assignin(sectionObj,entryName,entryValue)`

**Description**

`assignin(sectionObj,entryName,entryValue)` assigns the value `entryValue` to the data dictionary entry `entryName` in the data dictionary section `sectionObj`, a `Simulink.data.dictionary.Section` object. If an entry with the specified name is not in the target section, `assignin` creates the entry with the specified name and value.

If an entry with the name specified by input argument `entryName` is not defined in the target data dictionary section but is defined in a referenced dictionary, `assignin` does not create a new entry in the target section but operates on the entry in the referenced dictionary.

To programatically access variables for the purpose of sweeping block parameter values, consider using `Simulink.SimulationInput` objects instead of modifying the variables through the programmatic interface of the data dictionary. See “Optimize, Estimate, and Sweep Block Parameter Values”.

**Input Arguments**

- `sectionObj` — Target data dictionary section  
  Simulink.data.dictionary.Section object

Target data dictionary section, specified as a `Simulink.data.dictionary.Section` object. Before you use this function, represent the target section with a
Simulink.data.dictionary.Section object by using, for example, the getSection function.

**entryName — Name of target data dictionary entry**
character vector

Name of target data dictionary entry, specified as a character vector. If a matching entry does not already exist, the functions creates a new entry using the specified name.

Example: 'myEntry'
Data Types: char

**entryValue — Value to assign to data dictionary entry**
MATLAB expression

Value to assign to data dictionary entry, specified as a MATLAB expression that returns any valid data dictionary content.

Example: 27.5
Example: myBaseWorkspaceVariable
Example: Simulink.Parameter

## Examples

### Assign Value to Data Dictionary Entry

Assign a value to a data dictionary entry by operating on a Simulink.data.dictionary.Section object.

Represent the Design Data section of the data dictionary myDictionary_ex_API.sldd with a Simulink.data.dictionary.Section object named dDataSectObj.

```matlab
myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');
dDataSectObj = getSection(myDictionaryObj,'Design Data');
```

Assign the value 237 to an entry myAssignedEntry in the data dictionary myDictionary_ex_API.sldd. If an entry named myAssignedEntry is not in myDictionary_ex_API.sldd, create it.

```matlab
myAssignedEntry = Simulink.data.dictionary.Entry('Design Data', 'myAssignedEntry', 237);
```
assignin(dDataSectObj,'myAssignedEntry',237)

**Alternatives**

You can use the Model Explorer window to view and change the values of data dictionary entries.

**See Also**

Simulink.data.dictionary.Section | setValue

**Topics**

“Store Data in Dictionary Programmatically”

**Introduced in R2015a**
deleteEntry

Class: Simulink.data.dictionary.Section
Package: Simulink.data.dictionary

Delete data dictionary entry

Syntax

deleteEntry(sectionObj,entryName)
deleteEntry(sectionObj,entryName,'DataSource',dictionaryName)

Description

deleteEntry(sectionObj,entryName) deletes a data dictionary entry entryName from the data dictionary section sectionObj, a Simulink.data.dictionary.Section object. If there are multiple entries with the specified name in a hierarchy of reference dictionaries, the function deletes all the entries. If you represent a data dictionary entry with one or more Simulink.data.dictionary.Entry objects and later delete the entry using the deleteEntry function, the objects remain with their Status property set to 'Deleted'.

deleteEntry(sectionObj,entryName,'DataSource',dictionaryName) deletes an entry that is defined in the data dictionary DictionaryName. Use this syntax to uniquely identify an entry that is defined more than once in a hierarchy of referenced data dictionaries.

Input Arguments

sectionObj — Target data dictionary section
Simulink.data.dictionary_SECTION object

Target data dictionary section, specified as a Simulink.data.dictionary.Section object. Before you use this function, represent the target section with a
Simulink.data.dictionary.Section object by using, for example, the getSection function.

entryName — Name of target data dictionary entry  
character vector

Name of target data dictionary entry, specified as a character vector.
Example: 'myEntry'
Data Types: char

dictionaryName — Name of data dictionary that defines target entry  
character vector

File name of data dictionary that defines the target entry, specified as a character vector including the .sldd extension.
Example: 'mySubDictionary_ex_API.sldd'
Data Types: char

Examples

Delete Entry from Data Dictionary Section

Represent the Design Data section of the data dictionary myDictionary_ex_API.sldd with a Simulink.data.dictionary.Section object named dDataSectObj. The Design Data section of myDictionary_ex_API.sldd already contains an entry named fuelFlow.

myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');
dDataSectObj = getSection(myDictionaryObj,'Design Data');

Delete the entry fuelFlow from the data dictionary myDictionary_ex_API.sldd. myDictionary_ex_API.sldd no longer contains the fuelFlow entry.

deleteEntry(dDataSectObj,'fuelFlow')
Delete Entry from Reference Data Dictionary

Represent the Design Data section of the data dictionary myDictionary_ex.API.sldd with a Simulink.data.dictionary.Section object named dDataSectObj.

```
myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex.API.sldd');
dDataSectObj = getSection(myDictionaryObj,'Design Data');
```

Delete the entry myRefEntry from the data dictionary
myRefDictionary_ex.API.sldd. myRefDictionary_ex.API.sldd references
myRefDictionary_ex.API.sldd, and myRefDictionary_ex.API.sldd defines an entry myRefEntry.

```
deleteEntry(dDataSectObj,'myRefEntry','DataSource','...
'myRefDictionary_ex.API.sldd')
```

Alternatives

You can use the Model Explorer window to delete entries from a data dictionary in the same way you can delete variables from a model workspace or the base workspace.

See Also

Simulink.data.dictionary.Entry | Simulink.data.dictionary.Section | addEntry

Topics

“Store Data in Dictionary Programmatically”

Introduced in R2015a
evalin

Class: Simulink.data.dictionary.Section
Package: Simulink.data.dictionary

Evaluate MATLAB expression in data dictionary section

Syntax

returnValue = evalin(sectionObj,expression)

Description

returnValue = evalin(sectionObj,expression) evaluates a MATLAB expression in the data dictionary section sectionObj and returns the values returned by expression.

To programmatically access variables for the purpose of sweeping block parameter values, consider using Simulink.SimulationInput objects instead of modifying the variables through the programmatic interface of the data dictionary. See “Optimize, Estimate, and Sweep Block Parameter Values”.

Input Arguments

sectionObj — Target data dictionary section
Simulink.data.dictionary.Section object

Target data dictionary section, specified as a Simulink.data.dictionary.Section object. Before you use this function, represent the target section with a Simulink.data.dictionary.Section object by using, for example, the getSection function.

expression — MATLAB expression to evaluate
character vector

MATLAB expression to evaluate, specified as a character vector.
Examples

List All Entries in Data Dictionary Section

Represent the Design Data section of the data dictionary myDictionary_ex_API.sldd with a Simulink.data.dictionary.Section object named dDataSectObj.

myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');
dDataSectObj = getSection(myDictionaryObj,'Design Data');

Execute the whos command in the Design Data section of myDictionary_ex_API.sldd.

evalin(dDataSectObj,'whos')

<table>
<thead>
<tr>
<th>Name</th>
<th>Size</th>
<th>Bytes</th>
<th>Class</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>fuelFlow</td>
<td>1x1</td>
<td>8</td>
<td>double</td>
<td></td>
</tr>
<tr>
<td>myRefEntry</td>
<td>1x1</td>
<td>1</td>
<td>logical</td>
<td></td>
</tr>
<tr>
<td>parameterGain37</td>
<td>1x1</td>
<td>112</td>
<td>Simulink.Parameter</td>
<td></td>
</tr>
</tbody>
</table>

Tips

- evalin allows you to treat a data dictionary section as a MATLAB workspace. You can think of entries contained in the section as workspace variables you can manipulate with MATLAB expressions.

See Also

Simulink.data.dictionary.Section | Simulink.data.evalinGlobal

Topics

“Store Data in Dictionary Programmatically”
Introduced in R2015a
exist

**Class:** Simulink.data.dictionary.Section  
**Package:** Simulink.data.dictionary

Check existence of data dictionary entry

**Syntax**

doesExist = exist(sectionObj,entryName)

**Description**

doesExist = exist(sectionObj,entryName) determines if the data dictionary section sectionObj contains an entry by the name of entryName and returns an indication.

**Input Arguments**

- **sectionObj — Target data dictionary section**  
  Simulink.data.dictionary.Section object

  Target data dictionary section, specified as a Simulink.data.dictionary.Section object. Before you use this function, represent the target section with a Simulink.data.dictionary.Section object by using, for example, the getSection function.

- **entryName — Name of target entry**  
  character vector

  Name of target entry, specified as a character vector.

  Example: 'myEntry'

  Data Types: char
Output Arguments

doesExist — Indication of entry existence
0 | 1

Indication of entry existence, returned as 0 if false and 1 if true.

Examples

Determine if Data Dictionary Entry Exists

Determine if an entry exists in a data dictionary by searching for the name of the entry.

Represent the Design Data section of the data dictionary myDictionary_ex_API.sldd with a Simulink.data.dictionary.Section object named dDataSectObj.

myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');
dDataSectObj = getSection(myDictionaryObj,'Design Data');

Determine if an entry fuelFlow exists in the Design Data section of myDictionary_ex_API.sldd.

exist(dDataSectObj,'fuelFlow')

ans =

    1

Determine if an entry myEntry exists in the Design Data section of myDictionary_ex_API.sldd.

exist(dDataSectObj,'myEntry')
ans =

0

Tips

• `exist` also determines if a matching entry exists in the same section of any referenced data dictionaries. For example, if `sectionObj` represents the Design Data section of a data dictionary `myDictionary_ex_API.sldd`, `exist` searches the Design Data section of `myDictionary_ex_API.sldd` and the Design Data sections of any dictionaries referenced by `myDictionary_ex_API.sldd`.

Alternatives

You can use Model Explorer to search a data dictionary for an entry.

See Also

- `Simulink.data.dictionary.Section`
- `Simulink.data.existsInGlobal`
- `find`

Topics

- “Store Data in Dictionary Programmatically”

Introduced in R2015a
exportToFile

**Class:** Simulink.data.dictionary.Section  
**Package:** Simulink.data.dictionary

Export data dictionary entries from section to MAT-file or MATLAB file

**Syntax**

```matlab
exportToFile(sectionObj,fileName)
```

**Description**

`exportToFile(sectionObj,fileName)` exports to a MAT or MATLAB file all the values of the entries contained in the data dictionary section `sectionObj`, a Simulink.data.dictionary.Section object. `exportToFile` exports the values of all entries, including those defined in referenced dictionaries.

**Input Arguments**

- **sectionObj — Target data dictionary section**  
  `Simulink.data.dictionary.Section object`

  Target data dictionary section, specified as a `Simulink.data.dictionary.Section` object. Before you use this function, represent the target section with a `Simulink.data.dictionary.Section` object by using, for example, the `getSection` function.

- **fileName — Name of MAT or MATLAB file**  
  `character vector`

  Name of target MAT or MATLAB file, specified as a character vector. `exportToFile` supplies a file extension `.mat` if you do not specify an extension.

  Example: 'myNewFile.mat'
  Example: 'myNewFile.m'
Data Types: char

Examples

Export Data Dictionary Entries to MAT or MATLAB Files

Represent the Design Data section of the data dictionary myDictionary_ex_API.sldd with a Simulink.data.dictionary.Section object named dDataSectObj. Represent the Configurations section of myDictionary_ex_API.sldd with an object named configSectObj.

myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');
dDataSectObj = getSection(myDictionaryObj, 'Design Data');
configSectObj = getSection(myDictionaryObj, 'Configurations');

Export the entries from the Design Data section of myDictionary_ex_API.sldd to a MATLAB file in your current working folder.

exportToFile(dDataSectObj, 'myDictionaryDesignData.m')

Export the entries from the Configurations section of myDictionary_ex_API.sldd to a MAT-file in your current working folder.

exportToFile(configSectObj, 'myDictionaryConfigurations.mat')

Exported 1 entries from scope 'Configurations' to MAT-file myDictionaryConfigurations.mat.

Limitation

The exportToFile method does not export enumerated data types (which are stored as Simulink.data.dictionary.EnumTypeDefinition objects). To transfer or copy an enumerated type from one dictionary to another, use the getEntry and addEntry methods of Simulink.data.dictionary.Section objects.
Alternatives

You can use Model Explorer to export data dictionary entries to a file. See “Export Design Data from Dictionary” for more information.

See Also
Simulink.data.dictionary.Section | importFromFile

Topics
“Store Data in Dictionary Programmatically”

Introduced in R2015a
find

Class: Simulink.data.dictionary.Section
Package: Simulink.data.dictionary

Search in data dictionary section

Syntax

foundEntries = find(sectionObj,PName1,PValue1,...,PNameN,PValueN)
foundEntries = find(sectionObj,PName1,PValue1,...,PNameN,PValueN, options)

Description

foundEntries = find(sectionObj,PName1,PValue1,...,PNameN,PValueN) searches the data dictionary section sectionObj using search criteria PName1,PValue1,...,PNameN,PValueN, and returns an array of matching entries that were found in the target section. This syntax matches the search criteria with the properties of the entries in the target section but not with the properties of their values. See Simulink.data.dictionary.Entry for a list of data dictionary entry properties.

foundEntries = find(sectionObj,PName1,PValue1,...,PNameN,PValueN, options) searches for data dictionary entries using additional search options. For example, you can match the search criteria with the values of the entries in the target section.

Input Arguments

sectionObj — Data dictionary section to search
Simulink.data.dictionary.Section object

Data dictionary section to search, specified as a Simulink.data.dictionary.Section object. Before you use this function, represent the target section with a Simulink.data.dictionary.Section object by using, for example, the getSection function.
**PName1,PValue1,...,PNameN,PValueN — Search criteria**
name-value pairs representing properties

Search criteria, specified as one or more name-value pairs representing names and values of properties of the entries in the target data dictionary section. For a list of the properties of a data dictionary entry, see Simulink.data.dictionary.Entry. If you specify more than one name-value pair, the returned entries meet all of the criteria.

If you include the '-value' option to search in the values of the entries, the search criteria apply to the values of the entries rather than to the entries themselves.

Example: 'LastModifiedBy','jsmith'
Example: 'DataSource','myRefDictionary_ex_API.sldd'

**options — Additional search options**
supported option codes

Additional search options, specified as one or more of the following supported option codes.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'-value'</td>
<td>This option causes find to search only in the values of the entries in the target data dictionary section. Specify this option before any other search criteria or options arguments.</td>
</tr>
<tr>
<td>'-and','-or','-xor','-not'</td>
<td>These options modify or combine multiple search criteria or other option codes.</td>
</tr>
<tr>
<td>'-property',propertyName</td>
<td>This name-value pair causes find to search for entries or values that have the property propertyName regardless of the value of the property. Specify propertyName as a character vector.</td>
</tr>
<tr>
<td>'-class',className</td>
<td>This name-value pair causes find to search for entries or values that are objects of the class className. Specify className as a character vector.</td>
</tr>
<tr>
<td>'-isa',className</td>
<td>This name-value pair causes find to search for entries or values that are objects of the class or of any subclass derived from the class className. Specify className as a character vector.</td>
</tr>
</tbody>
</table>
This option allows you to use regular expressions in your search criteria. This option affects only search criteria that follow '-regexp'.

Example: '-value'
Example: '-value','-property','CoderInfo'
Example: '-value','-class','Simulink.Parameter'

Output Arguments

foundEntries — Data dictionary entries matching search criteria
array of Simulink.data.dictionary.Entry objects

Data dictionary entries matching the specified search criteria, returned as an array of Simulink.data.dictionary.Entry objects.

Examples

Return Array of All Entries in Data Dictionary Section

Represent the Design Data section of the data dictionary myDictionary_ex_API.sldd with a Simulink.data.dictionary.Section object named dDataSectObj.

myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');
dDataSectObj = getSection(myDictionaryObj,'Design Data');

Return all of the entries stored in the Design Data section of the data dictionary myDictionary_ex_API.sldd.

allEntries = find(dDataSectObj)

Search Data Dictionary Section for Specific Class

Represent the Design Data section of the data dictionary myDictionary_ex_API.sldd with a Simulink.data.dictionary.Section object named dDataSectObj.
myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');
dDataSectObj = getSection(myDictionaryObj,'Design Data');

Search in the Design Data section of myDictionary_ex_API.sldd for entries whose values are objects of the Simulink.Parameter class.

foundEntries = find(dDataSectObj,'-value','-class','Simulink.Parameter')

**Search Data Dictionary Section for Modifying User**

Represent the Design Data section of the data dictionary myDictionary_ex_API.sldd with a Simulink.data.dictionary.Section object named dDataSectObj.

myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');
dDataSectObj = getSection(myDictionaryObj,'Design Data');

Search in the Design Data section of myDictionary_ex_API.sldd for entries that were last modified by the user jsmith.

foundEntries = find(dDataSectObj,'LastModifiedBy','jsmith')

**Search Data Dictionary Section Using Multiple Criteria**

Represent the Design Data section of the data dictionary myDictionary_ex_API.sldd with a Simulink.data.dictionary.Section object named dDataSectObj.

myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');
dDataSectObj = getSection(myDictionaryObj,'Design Data');

Search in the Design Data section of myDictionary_ex_API.sldd for entries that were last modified by the user jsmith or whose names begin with fuel.

foundEntries = find(dDataSectObj,'LastModifiedBy','jsmith','-or',... '-regexp','Name','fuel*')

**Search Data Dictionary Section Using Regular Expressions**

Represent the Design Data section of the data dictionary myDictionary_ex_API.sldd with a Simulink.data.dictionary.Section object named dDataSectObj.
myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');
dDataSectObj = getSection(myDictionaryObj,'Design Data');

Search in the Design Data section of myDictionary_ex_API.sldd for entries whose names begin with fuel.
foundEntries = find(dDataSectObj,'-regexp', 'Name', 'fuel*')

Search Data Dictionary Section for Specific Value

Represent the Design Data section of the data dictionary myDictionary_ex_API.sldd with a Simulink.data.dictionary.Section object named dDataSectObj.

myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');
dDataSectObj = getSection(myDictionaryObj,'Design Data');

Return all of the entries stored in the Design Data section of the data dictionary myDictionary_ex_API.sldd.
allEntries = find(dDataSectObj);

Find the entries with value 237. If you find more than one entry, store the entries in an array called foundEntries.

foundEntries = [];
for i = 1:length(allEntries)
    if getValue(allEntries(i)) == 237
        foundEntries = [foundEntries allEntries(i)];
    end
end

Search Data Dictionary Section for Specific Property

Represent the Design Data section of the data dictionary myDictionary_ex_API.sldd with a Simulink.data.dictionary.Section object named dDataSectObj.

myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');
dDataSectObj = getSection(myDictionaryObj,'Design Data');

Search in the Design Data section of myDictionary_ex_API.sldd for entries whose values have a property DataType.
foundEntries = find(dDataSectObj,'-value','-property','DataType')

**Alternatives**

You can use Model Explorer to search a data dictionary for entries using arbitrary criteria.

**See Also**

Simulink.data.dictionary.Entry | Simulink.data.dictionary.Section | exist | find

**Topics**

“Store Data in Dictionary Programmatically”

**Introduced in R2015a**
**getEntry**

**Class:** Simulink.data.dictionary.Section  
**Package:** Simulink.data.dictionary

Create `Simulink.data.dictionary.Entry` object to represent data dictionary entry

**Syntax**

```matlab
entryObj = getEntry(sectionObj,entryName)
entryObj = getEntry(sectionObj,
  entryName,'DataSource',dictionaryName)
```

**Description**

`entryObj = getEntry(sectionObj,entryName)` returns an array of `Simulink.data.dictionary.Entry` objects representing data dictionary entries `entryName` found in the data dictionary section `sectionObj`, a `Simulink.data.dictionary.Section` object. `getEntry` returns multiple objects if multiple entries have the specified name in a reference hierarchy of data dictionaries.

`entryObj = getEntry(sectionObj,
  entryName,'DataSource',dictionaryName)` returns an object representing a data dictionary entry that is defined in the data dictionary `dictionaryName`. Use this syntax to uniquely identify an entry that is defined more than once in a hierarchy of referenced data dictionaries.

**Input Arguments**

- `sectionObj` — Target data dictionary section
  
  `Simulink.data.dictionary.Section` object

  Target data dictionary section, specified as a `Simulink.data.dictionary.Section` object. Before you use this function, represent the target section with a `Simulink.data.dictionary.Section` object by using, for example, the `getSection` function.
**entryName** — Name of target data dictionary entry
character vector

Name of target data dictionary entry, specified as a character vector.
Example: 'myEntry'
Data Types: char

**dictionaryName** — Name of data dictionary containing target entry
character vector

File name of data dictionary containing the target entry, specified as a character vector
including the .sldd extension.
Example: 'mySubDictionary_ex_API.sldd'
Data Types: char

### Output Arguments

**entryObj** — Target data dictionary entry
Simulink.data.dictionary.Entry object

Target data dictionary entry, returned as one or more
Simulink.data.dictionary.Entry objects.

### Examples

#### Set Value of Data Dictionary Entry

Represent the Design Data section of the data dictionary myDictionary_ex_API.sldd
with a Simulink.data.dictionary.Section object named dDataSectObj.

```
myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');
dDataSectObj = getSection(myDictionaryObj,'Design Data');
```

Represent the data dictionary entry fuelFlow with a
Simulink.data.dictionary.Entry object named fuelFlowObj. fuelFlow is
defined in the data dictionary myDictionary_ex_API.sldd.
fuelFlowObj = getEntry(dDataSectObj,'fuelFlow');
Set the value of the entry fuelFlow to 493.
setValue(fuelFlowObj,493)

Set Value of Entry in Reference Dictionary

Represent the Design Data section of the data dictionary myDictionary_ex_API.sldd with a Simulink.data.dictionary.Section object named dDataSectObj.

myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');
dDataSectObj = getSection(myDictionaryObj,'Design Data');

Represent the data dictionary entry myRefEntry with a Simulink.data.dictionary.Entry object named refEntryObj.
myDictionary_ex_API.sldd references myRefDictionary_ex_API.sldd, and myRefDictionary_ex_API.sldd defines an entry myRefEntry.

refEntryObj = getEntry(dDataSectObj,'myRefEntry','DataSource',... 'myRefDictionary_ex_API.sldd');
Set the value of the entry myRefEntry to 493.
setValue(refEntryObj,493)

See Also
Simulink.data.dictionary.Entry | Simulink.data.dictionary.Section | addEntry | getValue

Topics
“Store Data in Dictionary Programmatically”

Introduced in R2015a
importFromFile

**Class:** Simulink.data.dictionary.Section  
**Package:** Simulink.data.dictionary

Import variables from MAT-file or MATLAB file to data dictionary section

**Syntax**

```matlab
importedVars = importFromFile(sectionObj,fileName)
importedVars = importFromFile(sectionObj,fileName,'existingVarsAction',existAction)
[importedVars,existingVars] = importFromFile(____)
```

**Description**

`importedVars = importFromFile(sectionObj,fileName)` imports variables defined in the MAT-file or MATLAB file `fileName` to the data dictionary section `sectionObj` without overwriting any variables that are already in the target section. If any variables are already in the target section, the function displays a warning and a list in the MATLAB Command Window. This syntax returns a list of variables that were successfully imported. A variable is considered successfully imported only if `importFromFile` assigns the value of the variable to the corresponding entry in the target data dictionary.

`importedVars = importFromFile(sectionObj,fileName,'existingVarsAction',existAction)` imports variables that are already in the target section by taking a specified action `existAction`. For example, you can choose to use the variable values in the target file to overwrite the corresponding values in the target section.

`[importedVars,existingVars] = importFromFile(____)` returns a list of variables in the target section that were not overwritten. Use this syntax if `existingVarsAction` is set to 'none', the default value, which prevents existing dictionary entries from being overwritten.
Input Arguments

**sectionObj — Target data dictionary section**

*Simulink.data.dictionary.Section* object

Target data dictionary section, specified as a *Simulink.data.dictionary.Section* object. Before you use this function, represent the target section with a *Simulink.data.dictionary.Section* object by using, for example, the *getSection* function.

**fileName — Name of MAT or MATLAB file**

*character vector*

Name of target MAT or MATLAB file, specified as a character vector. *importFromFile* automatically supplies a file extension .mat if you do not specify an extension.

Example: 'myFile.mat'
Example: 'myFile.m'
Data Types: char

**existAction — Action to take for existing dictionary variables**

'none' (default) | 'overwrite' | 'error'

Action to take for existing dictionary variables, specified as 'none', 'overwrite', or 'error'.

If you specify 'none', *importFromFile* attempts to import target variables but does not import or make any changes to variables that are already in the data dictionary section.

If you specify 'overwrite', *importFromFile* imports all target variables and overwrites any variables that are already in the data dictionary section.

If you specify 'error', *importFromFile* returns an error, without importing any variables, if any target variables are already in the data dictionary section.

Example: 'overwrite'
Data Types: char
Output Arguments

`importedVars` — Successfully imported variables

cell array of character vectors

Names of successfully imported variables, returned as a cell array of character vectors. A variable is considered successfully imported only if `importFromFile` assigns its value to the corresponding entry in the target data dictionary.

`existingVars` — Variables that were not imported

cell array of character vectors

Names of target variables that were not imported due to their existence in the target data dictionary, returned as a cell array of character vectors. `existingVars` has content only if `existAction` is set to 'none', which is also the default. In that case `importFromFile` imports only variables that are not already in the target data dictionary.

Examples

Import to Data Dictionary from File

Represent the Design Data section of the data dictionary `myDictionary_ex_API.sldd` with a `Simulink.data.dictionary.Section` object named `dDataSectObj`.

```matlab
myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');
dDataSectObj = getSection(myDictionaryObj,'Design Data');
```

Import all variables contained in the file `myData_ex_API.m` to the data dictionary and return a list of successfully imported variables. If any variables are already in `myDictionary_ex_API.sldd`, `importFromFile` returns a warning and a list of the affected variables.

```matlab
importFromFile(dDataSectObj,'myData_ex_API.m')
```

Warning: The following variables were not imported because they already exist in the dictionary:

```
   fuelFlow
```

```matlab
ans =
```
Import Variables from File and Overwrite Conflicts

Represent the Design Data section of the data dictionary myDictionary_ex_API.sldd with a Simulink.data.dictionary.Section object named dDataSectObj.

```matlab
myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');
dDataSectObj = getSection(myDictionaryObj,'Design Data');
```

Import all variables contained in the file myData_ex_API.m to the data dictionary, overwrite any variables that are already in the dictionary, and return a list of successfully imported variables.

```matlab
importFromFile(dDataSectObj,'myData_ex_API.m','existingVarsAction','overwrite')
ans =
    'fuelFlow'
    'myFirstEntry'
    'mySecondEntry'
    'myThirdEntry'
```

Return Variables Not Imported to Data Dictionary from File

Return a list of variables that are not imported from a file because they are already in the target data dictionary

Represent the Design Data section of the data dictionary myDictionary_ex_API.sldd with a Simulink.data.dictionary.Section object named dDataSectObj.

```matlab
myDictionaryObj = Simulink.data.dictionary.open('myDictionary_ex_API.sldd');
dDataSectObj = getSection(myDictionaryObj,'Design Data');
```

Import all variables contained in the file myData_ex_API.m to the data dictionary. Specify names for the output arguments of importFromFile to return the names of successfully and unsuccessfully imported variables.
[importedVars,existingVars] = importFromFile(dDataSectObj,'myData_ex_API.m')

importedVars =
    'myFirstEntry'
    'mySecondEntry'
    'myThirdEntry'

existingVars =
    'fuelFlow'

importFromFile does not import the variable fuelFlow because it is already in the target data dictionary.

**Tips**

- `importFromFile` can import MATLAB variables created from enumerated data types but cannot import the definitions of the enumerated types. Use the `importEnumTypes` function to import enumerated data type definitions to a data dictionary. If you import variables of enumerated data types to a data dictionary but do not import the enumerated type definitions, the dictionary is less portable and might not function properly if used by someone else.

**Alternatives**

You can use the Model Explorer to import variables to a data dictionary from a file. See “Import Data to Dictionary from File” for more information.

**See Also**

Simulink.data.dictionary.Section | exportToFile | importEnumTypes

**Topics**

“Store Data in Dictionary Programmatically”
Introduced in R2015a
Simulink.DualScaledParameter

Specify name, value, units, and other properties of Simulink dual-scaled parameter

Description

Use Simulink.DualScaledParameter so that you can define an object that stores two scaled values of the same physical value.

For example, for temperature measurement, you can store a Fahrenheit scale and a Celsius scale with conversion defined by a computation method that you provide. Given one scaled value, the Simulink.DualScaledParameter computes the other scaled value using the computation method.

A dual-scaled parameter inherits some properties from the Simulink.Parameter class. A dual-scaled parameter has:

- A calibration value. The value that you prefer to use.
- A main value. The real-world value that Simulink uses.
- An internal stored integer value. The value that is used in the embedded code.

You can use Simulink.DualScaledParameter objects in your model for both simulation and code generation. The parameter computes the internal value before code generation via the computation method. This offline computation results in leaner generated code.

If you provide the calibration value, the parameter computes the main value using the computation method. This method can be a first-order rational function.

\[ y = \frac{ax + b}{cx + d} \]

- \( x \) is the calibration value.
- \( y \) is the main value.
- \( a \) and \( b \) are the coefficients of the CalToMain compute numerator.
- \( c \) and \( d \) are the coefficients of the CalToMain compute denominator.
If you provide the calibration minimum and maximum values, the parameter computes minimum and maximum values of the main value. Simulink performs range checking of parameter values. The software alerts you when the parameter object value lies outside a range that corresponds to its specified minimum and maximum values and data type.

**Creation**

Create a `Simulink.DualScaledParameter` object:

- By using the Model Data Editor. Instead of creating a `Simulink.Parameter` object, create a `Simulink.DualScaledParameter` object. See “Interact with a Model That Uses Workspace Variables”.
- By using the Model Explorer:
  1. In the **Model Hierarchy** pane, select a workspace or data dictionary.
  2. On the toolbar, select **Add > Add Custom**.
  3. In the Model Explorer — Select Object dialog box, set **Object class** to `Simulink.DualScaledParameter`.
- By using the `Simulink.DualScaledParameter` function, described below.

**Syntax**

```matlab
DSParam = Simulink.DualScaledParameter
```

**Description**

`DSParam = Simulink.DualScaledParameter` returns a `Simulink.DualScaledParameter` object with default property values.

**Properties**

For information about properties in the property dialog box of a `Simulink.DualScaledParameter` object, see “Simulink.DualScaledParameter Property Dialog Box”.

5-1050
**CalibrationValue — Calibration value of this parameter**

[ ] (default) | finite, real, double number

Calibration value of this parameter, specified as a finite, real, double number. This value represents the value that you prefer to use.

Before specifying CalibrationValue, you must specify CalToMainCompuNumerator and CalToMainCompuDenominator to define the computation method. The parameter uses the computation method and the calibration value to calculate the main value that Simulink uses.

Corresponds to **Calibration value** in the property dialog box.

Example: 5.34

Data Types: double

**CalibrationMin — Calibration minimum value of this parameter**

[ ] (default) | finite, real, double, scalar number

Calibration minimum value of this parameter, specified as a finite, real, double, scalar number. The default value, [], means the minimum is unspecified.

Before specifying CalibrationMin, you must specify CalToMainCompuNumerator and CalToMainCompuDenominator to define the computation method. The parameter uses the computation method and the calibration minimum value to calculate the minimum or maximum value that Simulink uses. A first order rational function is strictly monotonic, either increasing or decreasing. If it is increasing, setting the calibration minimum sets the main minimum value. If it is decreasing, setting the calibration minimum sets the main maximum.

If the parameter value is less than the minimum value or if the minimum value is outside the range of the parameter data type, Simulink generates a warning. In these cases, when updating the diagram or starting a simulation, Simulink generates an error.

Corresponds to **Calibration minimum** in the property dialog box.

Example: 10.51

Data Types: double

**CalibrationMax — Calibration maximum value of this parameter**

[ ] (default) | finite, real, double, scalar number
Calibration maximum value of this parameter, specified as a finite, real, double, scalar number. The default value, [ ], means the maximum is unspecified.

Before specifying CalibrationMax, you must specify CalToMainCompuNumerator and CalToMainCompuDenominator to define the computation method. The parameter uses the computation method and the calibration maximum value to calculate the corresponding maximum or minimum value that Simulink uses. A first order rational function is strictly monotonic, either increasing or decreasing. If it is increasing, setting the calibration maximum sets the main maximum value. If it is decreasing, setting the calibration maximum sets the main minimum.

If the parameter value is less than the minimum value or if the minimum value is outside the range of the parameter data type, Simulink generates a warning. In these cases, when updating the diagram or starting a simulation, Simulink generates an error.

Corresponds to Calibration maximum in the property dialog box.

Example: -10.51

Data Types: double

CalToMainCompuNumerator — Numerator coefficients of the computation method

[ ] (default) | finite, real, double scalar | finite, real, double vector

Numerator coefficients of the computation method, specified as a scalar number or vector of values for the numerator coefficients a and b of the first-order linear equation:

\[ y = \frac{ax + b}{cx + d} \]

The default value is [ ] (unspecified). Specify finite, real, double scalar values for a and b. For example, [1 1] or, for reciprocal scaling, 1.

Once you have applied CalToMainCompuNumerator, you cannot change it.

Corresponds to CalToMain compute numerator in the property dialog box.

Example: [1 1]

Example: 1

Data Types: double
CalToMainCompuDenominator — Denominator coefficients of the computation method

[] (default) | finite, real, double scalar | finite, real, double vector

Denominator coefficients of the computation method, specified as a scalar number or vector of values for the denominator coefficients $c$ and $d$ of the first-order linear equation:

$$y = \frac{ax + b}{cx + d}$$

The default value is [] (unspecified). Specify finite, real, double scalar values for $c$ and $d$. For example, [1 1].

Once you have applied CalToMainCompuDenominator, you cannot change it.

Corresponds to CalToMain compute denominator in the property dialog box.

Example: [1 1]
Data Types: double

CalibrationName — Name of the calibration parameter

'' (empty character vector) (default) | character vector

Name of the calibration parameter, specified as a character vector.

Corresponds to Calibration name in the property dialog box.

Example: 'This is a calibration parameter.'
Data Types: char

CalibrationDocUnits — Measurement units for this calibration parameter's value

'' (empty character vector) (default) | character vector

Measurement units for this calibration parameter's value, specified as a character vector.

Corresponds to Calibration units in the property dialog box.

Example: 'Fahrenheit'
Data Types: char

IsConfigurationValid — Information about validity of configuration

true (default) | false
This property is read-only.

Information about the validity of the object configuration, returned as true (valid) or false (invalid). If Simulink detects an issue with the configuration, it sets this field to false and provides information in the DiagnosticMessage property.

Corresponds to Is configuration valid in the property dialog box.

Data Types: logical

**DiagnosticMessage** — Diagnostic information about invalid configuration

' ' (empty character vector) (default) | character vector

This property is read-only.

Diagnostic information about an invalid object configuration, returned as a character vector. If you specify invalid property settings, Simulink displays a message in this field. Use the diagnostic information to help you fix an invalid configuration issue.

Corresponds to Diagnostic message in the property dialog box.

Data Types: char

**Examples**

**Create and Update a Dual-Scaled Parameter**

Create a Simulink.DualScaledParameter object that stores a temperature as both Fahrenheit and Celsius.

Create a `Simulink.DualScaledParameter` object.

```matlab
Temp = Simulink.DualScaledParameter;
```

Set the computation method that converts between Fahrenheit and Celsius.

```matlab
Temp.CalToMainCompuNumerator = [1 -32];
Temp.CalToMainCompuDenominator = [1.8];
```

Set the value of the temperature that you want to see in Fahrenheit.

```matlab
Temp.CalibrationValue = 212
```
Temp =

    DualScaledParameter with properties:

        CalibrationValue: 212
        CalibrationMin: []
        CalibrationMax: []
        CalToMainCompuNumerator: [1 -32]
        CalToMainCompuDenominator: 1.8000
        CalibrationName: ''
        CalibrationDocUnits: ''
        IsConfigurationValid: 1
        DiagnosticMessage: ''
        Value: 100
        CoderInfo: [1x1 Simulink.CoderInfo]
        Description: ''
        DataType: 'auto'
        Min: []
        Max: []
        Unit: ''
        Complexity: 'real'
        Dimensions: [1 1]

The Simulink.DualScaledParameter calculates Temp.Value which is the value that Simulink uses. Temp.CalibrationValue is 212 (degrees Fahrenheit), so Temp.Value is 100 (degrees Celsius).

Name the value and specify the units.

Temp.CalibrationName = 'TempF';
Temp.CalibrationDocUnits = 'Fahrenheit';

Set calibration minimum and maximum values.

Temp.CalibrationMin = 0;
Temp.CalibrationMax = 300;

If you specify a calibration value outside this allowable range, Simulink generates a warning.

Specify the units that Simulink uses.

Temp.Unit = 'degC';

Open the Simulink.DualScaledParameter dialog box.
open Temp
The **Calibration Attributes** tab displays the calibration value and the computation method that you specified.

In the dialog box, click the **Main Attributes** tab.
This tab displays information about the value used by Simulink.

**See Also**
AUTOSAR.DualScaledParameter | Simulink.Parameter

**Topics**
“Configure AUTOSAR Data for Measurement and Calibration” (AUTOSAR Blockset)
“Fixed Point”

**Introduced in R2013b**
Simulink.Mask class

Package: Simulink

Control masks programmatically

Description

Use an instance of Simulink.Mask class to perform the following operations:

- Create, copy, and delete masks.
- Create, edit, and delete mask parameters.
- Determine the block that owns the mask.
- Obtain workspace variables defined for a mask.

Properties

Type

Specifies the mask type of the associated block.

Type: character vector | string scalar

Default: Empty character vector

Description

Specifies the block description of the associated block.

Type: character vector | string scalar

Default: Empty character vector

Help

Specifies the help text that is displayed for the mask.
**Type:** character vector | string scalar

**Default:** Empty character vector

**Initialization**
Specifies the initialization commands for the associated block.

**Type:** character vector | string scalar

**Default:** Empty character vector

**SelfModifiable**
Indicates that the block can modify itself and its content.

**Type:** boolean

**Values:** 'on' | 'off'

**Default:** 'off'

**Display**
Specifies MATLAB code for drawing the block icon.

**Type:** character vector | string scalar

**Default:** Empty character vector

**IconFrame**
Sets the visibility of the block frame. (Visible is on, Invisible is off).

**Type:** boolean

**Values:** 'on' | 'off'

**Default:** 'on'

**IconOpaque**
Sets the transparency of the icon (Opaque is on, Transparent is off).
Type: boolean
Values: 'on' | 'off'
Default: 'on'

RunInitForIconRedraw
Specifies whether Simulink must run mask initialization before executing the mask icon commands.
Type: enum
Values: 'auto' | 'on' | 'off'
Default: 'auto'

IconRotate
Sets icon to rotate with the block.
Type: enum
Values: 'none' | 'port'
Default: 'none'

PortRotate
Specifies the port rotation policy for the masked block.
Type: enum
Values: 'default' | 'physical'
Default: 'default'

IconUnits
Specifies the units for the drawing commands.
Type: enum
Values: 'pixel' | 'autoscale' | 'normalized'
Default: 'autoscale'
Methods

addParameter Add a parameter to a mask
copy Copy a mask from one block to another
create Create a mask on a Simulink block
delete Unmask a block and delete the mask from memory
get Get a block mask as a mask object
addDialogControl Add dialog control elements to mask dialog box
getDialogControl Search for a specific dialog control on the mask
getOwner Determine the block that owns a mask
getParameter Get a mask parameter using its name
getWorkspaceVariables Get all the variables defined in the mask workspace for a masked block
numParameters Determine the number of parameters in a mask
removeDialogControl Remove dialog control element from mask dialog box
removeParameter Remove parameter from mask dialog box
removeAllParameters Remove all existing parameters from a mask
set Set the properties of an existing mask
addParameterConstraint Add parameter constraint to a mask
removeParameterConstraint Delete a mask parameter constraint
removeCrossParameterConstraint Delete a cross-parameter constraint
removeAllParameterConstraints Delete all mask parameter constraints
removeAllCrossParameterConstraints Delete all cross-parameter constraints from a mask
getParameterConstraint Get mask parameter constraint properties
crossParameterConstraint Get cross-parameter constraint
getAssociatedParametersOfConstraint Get mask parameters associated with a constraint
addCrossParameterConstraint Add cross-parameter constraint
See Also

Topics
“Control Masks Programmatically”
“Create Block Masks”
addParameter

Class: Simulink.Mask
Package: Simulink

Add a parameter to a mask

Syntax

p = Simulink.Mask.get(blockName)
p.addParameter(Name,Value)

Description

p = Simulink.Mask.get(blockName) gets the mask on the block specified by blockName as a mask object.

p.addParameter(Name,Value) appends a parameter to the mask. If you do not specify name-value pairs as arguments with this command, Simulink generates name for the mask parameter with control type set to edit.

Input Arguments

blockName

The handle to the block or the path to the block inside the model.

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside quotes. You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.
**Type**

Type of control that is used to specify the value of this parameter. The permitted values are: edit, checkbox, popup, combobox, listbox, radiobutton, slider, dial, spinbox, unit, textarea, breakpoint, customtable, tree, unit, min, max.

**Default:** edit

**TypeOptions**

The options that are displayed within a popup control or in a promoted parameter. This field is a cell array.

**Default:** empty

**Name**

The name of the mask parameter. This name is assigned to the mask workspace variable created for this parameter.

**Default:** generated name

**Prompt**

Text that identifies the parameter on the Mask Parameters dialog box.

**Default:** empty

**Value**

The default value of the mask parameter in the Mask Parameters dialog box.

**Default:** Type specific; depends on the Type of the parameter

**Evaluate**

Option to specify whether parameter must be evaluated.

**Default:** 'on'

**Tunable**

Option to specify whether parameter is tunable.
Default: 'on'

**Enabled**

Option to specify whether user can set parameter value.

Default: 'on'

**Visible**

Option to set whether mask parameter is hidden or visible to the user.

Default: 'on'

**Callback**

Container for MATLAB code that executes when user makes a change in the Mask Parameters dialog box and clicks **Apply**.

Default: empty

**Container**

Option to specify a container for the child parameters. The permitted values are the names of 'panel', 'group', and 'tab'.

**Examples**

1. Get mask as an object using a masked block's path.
   
   ```matlab
   p = Simulink.Mask.get('myModel/Subsystem');
   ```

2. Add a parameter to the mask without specifying name-value pairs for parameter attributes.
   
   ```matlab
   p.addParameter;
   ```

3. Add a mask parameter of type popup that cannot be evaluated.
   
   ```matlab
   p.addParameter('Type','popup','TypeOptions',{ 'Red' 'Blue' 'Green'},'Evaluate','off');
   ```
See Also

“Create Block Masks” | Simulink.Mask
Simulink.Mask.copy

Class: Simulink.Mask
Package: Simulink

Copy a mask from one block to another

Syntax

pSource = Simulink.Mask.get(srcBlockName)
pDest = Simulink.Mask.create(destBlockName)
pDest.copy(pSource)

Description

pSource = Simulink.Mask.get(srcBlockName) gets the mask on the source block specified by blockName as a mask object.

pDest = Simulink.Mask.create(destBlockName) creates an empty mask on the destination block specified by destBlockName.

pDest.copy(pSource) overwrites the destination mask with the source mask.

Input Arguments

srcBlockName

The handle to the source block or the path to the source block inside the model.

Note The source block should be masked.

destBlockName

The handle to the destination block or the path to the destination block inside the model.
Note The destination block should have an empty mask. Otherwise, the copied mask will overwrite the non-empty mask.

Examples

1  Create an empty mask on the destination block using the block’s path.
   pDest = Simulink.Mask.create('myModel/Subsystem');
2  Get source mask as an object using the source block’s path.
   pSource = Simulink.Mask.get('myModel/Abs');
3  Make the destination mask a copy of the source mask.
   pDest.copy(pSource);

See Also
“Create Block Masks” | Simulink.Mask
Simulink.Mask.create

Class: Simulink.Mask  
Package: Simulink

Create a mask on a Simulink block

Syntax

\[ p = \text{Simulink.Mask.create}(\text{blockName}) \]

Description

\[ p = \text{Simulink.Mask.create}(\text{blockName}) \] creates an empty mask on the block specified by \text{blockName}. If the specified block is already masked, an error message appears.

Input Arguments

\text{blockName}

The handle to the block or the path to the block inside the model.

Examples

1. Create a mask using a block’s handle.

\begin{verbatim}
   Note In the model, select the block to be masked.

   p = Simulink.Mask.create(gcah);
\end{verbatim}

2. Create a mask using the block’s path.

\begin{verbatim}
   p = Simulink.Mask.create('myModel/Subsystem');
\end{verbatim}
See Also
“Create Block Masks” | Simulink.Mask
delete

**Class:** Simulink.Mask  
**Package:** Simulink

Unmask a block and delete the mask from memory

**Syntax**

```matlab
p = Simulink.Mask.get(blockName)
p.delete
```

**Description**

`p = Simulink.Mask.get(blockName)` gets the mask on the block specified by `blockName` as a mask object.

`p.delete` unmask the block and deletes the mask from memory.

**Input Arguments**

`blockName`

The handle to the block or the path to the block inside the model.

**Examples**

1. Get mask as an object using a masked block's path.

   ```matlab
   p = Simulink.Mask.get('myModel/Subsystem');
   ```

2. Unmask the block using the mask object and delete the mask from memory.

   ```matlab
   p.delete;
   ```
See Also
“Create Block Masks” | Simulink.Mask
Simulink.Mask.get

**Class:** Simulink.Mask  
**Package:** Simulink

Get a block mask as a mask object

**Syntax**

```
p = Simulink.Mask.get(blockName)
```

**Description**

```
p = Simulink.Mask.get(blockName)  
gets the mask on the block specified by blockName as a mask object. If the specified block is not masked, a null value returns.
```

**Input Arguments**

**blockName**

The handle to the block or the path to the block inside the model.

**Examples**

1. Get mask as an object using a masked block's handle.

   **Note** In the model, select the masked block.

   ```
p = Simulink.Mask.get(gcbh);
```

2. Get mask as an object using a masked block's path.

   ```
p = Simulink.Mask.get('myModel/Subsystem');
```
See Also
“Create Block Masks” | Simulink.Mask
addDialogControl

**Class:** Simulink.Mask
**Package:** Simulink

Add dialog control elements to mask dialog box

**Syntax**

```matlab
successIndicator = maskObj.addDialogControl(controlType, controlIdentifier)
successIndicator = maskObj.addDialogControl(Name,Value)
```

**Description**

```matlab
successIndicator = maskObj.addDialogControl(controlType, controlIdentifier)
```
adds dialog control elements like text, hyperlinks, or tabs to mask dialog box. First get the mask object and assign it to the variable `maskObj`.

```matlab
successIndicator = maskObj.addDialogControl(Name,Value)
```
specifies the Name and Value arguments for an element on the mask dialog box. You can specify multiple Name-Value pairs.

**Input Arguments**

- `controlType` — Value type of dialog control element
  
  character vector | string scalar

  Type of dialog control element, specified

  - `'panel'`
  - `'group'`
  - `'tabcontainer'`
  - `'tab'`
• 'collapsiblepanel'
• 'text'
• 'image'
• 'hyperlink'
• 'pushbutton'

**controlIdentifier — Unique identifier for the element**
character vector | string scalar

Specifies the programmatic identifier for the element of mask dialog box. Use a name that is unique and does not have space between words. For more information, see “Variable Names” (MATLAB).

**Name-Value Pair Arguments**

Specify optional comma-separated pairs of Name,Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside single quotes (' ') and is case-sensitive. You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

**Type**

Type of control that is used to specify the value of this dialog control element. Type is a required argument. The permitted values are 'panel', 'group', 'tabcontainer', 'tab', 'collapsiblepanel', 'text', 'image', 'hyperlink', and 'pushbutton'.

**Default:** empty

**Name**

The identifier of the dialog control element. Name is a required argument. This field is available for all dialog control types.

**Default:** empty

**Prompt**

Text that is displayed in the dialog control element on the Mask dialog box. This field is available for all except for panel and image dialog control types.

**Default:** empty
**Tooltip**

Tooltip for the dialog control element.

**Default:** empty

**Enabled**

Option to specify whether you can set value for the dialog control element. This field is available for all dialog control types.

**Default:** 'on'

**Visible**

Option to set whether the dialog control element is hidden or visible to the user. This field is available for all dialog control types.

**Default:** 'on'

**Callback**

Container for MATLAB code that executes when you edit the dialog control element and click **Apply**. This field is available only for the hyperlink and pushbutton dialog control types.

**Default:** empty

**Row**

Option to set whether the dialog control is placed in the new row or the same row. This field is available for all dialog control types.

**Default:** empty

**Filepath**

Contains the path to an image file. This field is available for image, and pushbutton dialog control types.

**Default:** empty
**Container**

Option to specify a container for the child dialog control. The permitted values are the names of 'panel', 'group', and 'tab' dialog controls.

**Examples**

**Add Dialog Control Elements to Mask Dialog Box**

Get mask object and add dialog control element to it.

```matlab
% Get mask object on model Engine
maskObj = Simulink.Mask.get('Engine/Gain');

% Add hyperlink to mask dialog box
maskLink = maskObj.addDialogControl('hyperlink','link');
maskLink.Prompt = 'Mathworks Home Page';
maskLink.Callback = 'web(''www.mathworks.com'')'

% Alternative method to add hyperlink
maskLink = maskObj.addDialogControl('hyperlink','link');
maskLink.Prompt = 'www.mathworks.com';

% Add text to mask dialog box
maskText = maskObj.addDialogControl('text','text_tag');
maskText.Prompt = 'Enable range checking';

% Add button to mask dialog box
maskButton = maskObj.addDialogControl('pushbutton','button_tag');
maskButton.Prompt = 'Compute';
```

**Add Dialog Control Elements to Mask Dialog Box Tabs**

Create tabs on the mask dialog box and add elements to these tabs.
% Get mask object on a block named 'GainBlock'
maskObj = Simulink.Mask.get('GainBlock/Gain');

% Create a tab container
maskObj.addDialogControl('tabcontainer','allTabs');
tabs = maskObj.getDialogControl('allTabs');

% Create tabs and name them
maskTab1 = tabs.addDialogControl('tab','First');
maskTab1.Prompt = 'First tab';

maskTab2 = tabs.addDialogControl('tab','Second');
maskTab2.Prompt = 'Second tab';

% Add elements to one of the tabs
firstTab = tabs.getDialogControl('First');
firstTab.addDialogControl('text','textOnFirst');
firstTab.getDialogControl('textOnFirst').Prompt = 'Tab one';

**Add Dialog Control Element Using Name-Value Pair**

Add dialog control element and specify values for it

% Get mask object on model Engine
maskObj = Simulink.Mask.get('Engine/Gain');

% Add a dialog box and specify values for it
maskDialog = maskObj.addDialogControl('Type','text',...
   'Prompt','hello','Visible','off');

**See Also**

“Create Block Masks” | Control Masks Programmatically | Simulink.Mask

*Introduced in R2014a*
getDialogControl

**Class:** Simulink.Mask

**Package:** Simulink

Search for a specific dialog control on the mask

**Syntax**

\[ \text{[control, phandle]} = \text{handle.getDialogControl(cname)} \]

**Description**

\[ \text{[control, phandle]} = \text{handle.getDialogControl(cname)}, \text{search for a specific child dialog control recursively on the mask dialog.} \]

**Input Arguments**

cname

Name of the dialog control being searched on the mask dialog.

**Default:**

**Output Arguments**

control

Target dialog control being searched on the mask dialog.

phandle

Parent of the dialog control being searched mask dialog.
Examples

Find a dialog control

Find a text dialog control on the mask dialog box. maskObj is the handle to the mask object. The getDialogControl method returns the handle to the dialog control (hdlgctrl) and handle to the parent dialog control (phandle).

[hdlgctrl, phandle] = maskObj.getDialogControl('txt_var')

See Also
“Create Block Masks” | Simulink.Mask
**getOwner**

**Class:** Simulink.Mask  
**Package:** Simulink

Determine the block that owns a mask

**Syntax**

```plaintext
p = Simulink.Mask.get(blockName)
p.getOwner
```

**Description**

`p = Simulink.Mask.get(blockName)` gets the mask on the block specified by `blockName` as a mask object.

`p.getOwner` returns the interface to the block that owns the mask.

**Input Arguments**

`blockName`

The handle to the block or the path to the block inside the model.

**Examples**

1. Get mask as an object using a masked block's path.
   ```plaintext
   p = Simulink.Mask.get('myModel/Subsystem');
   ```
2. Get the interface to the block that owns the mask.
   ```plaintext
   p.getOwner;
   ```
See Also
“Create Block Masks” | Simulink.Mask
getParameter

Class: Simulink.Mask
Package: Simulink

Get a mask parameter using its name

Syntax

\[
p = \text{Simulink.Mask.get(blockName)}
\]
\[
\text{param} = p.\text{getParameter(paramName)}
\]

Description

\[
p = \text{Simulink.Mask.get(blockName)} \text{ gets the mask on the block specified by blockName as a mask object.}
\]
\[
\text{param} = p.\text{getParameter(paramName)} \text{ returns an array of mask parameters.}
\]

Input Arguments

blockName

The handle to the block or the path to the block inside the model.

paramName

The name of the parameter you want to get.

Examples

1. Get mask as an object using a masked block’s path.

\[
p = \text{Simulink.Mask.get('myModel/Subsystem')};
\]
2   Get a mask parameter by using its name.
    
    param = p.getParameter('intercept');

See Also
“Create Block Masks” | Simulink.Mask
getWorkspaceVariables

**Class:** Simulink.Mask  
**Package:** Simulink

Get all the variables defined in the mask workspace for a masked block

**Syntax**

```matlab
p = Simulink.Mask.get(blockName)
vars = p.getWorkspaceVariables
```

**Description**

- `p = Simulink.Mask.get(blockName)` gets the mask on the block specified by `blockName` as a mask object.

- `vars = p.getWorkspaceVariables` returns as a structure all the variables defined in the mask workspace for the masked block.

**Input Arguments**

- `blockName`  
The handle to the block or the path to the block inside the model.

**Examples**

1. Get mask as an object using a masked block's path.

   ```matlab
   p = Simulink.Mask.get('myModel/Subsystem');
   ```

2. Get all the variables defined in the mask workspace for the masked block.

   ```matlab
   vars = p.getWorkspaceVariables;
   ```
See Also

“Create Block Masks” | Simulink.Mask
numParameters

Class: Simulink.Mask
Package: Simulink

Determine the number of parameters in a mask

Syntax

\[
p = \text{Simulink.Mask.get(blockName)}
p.numParameters
\]

Description

\[ p = \text{Simulink.Mask.get(blockName)} \] gets the mask on the block specified by blockName as a mask object.

\[ p.numParameters \] returns the number of parameters in the mask.

Input Arguments

blockName

The handle to the block or the path to the block inside the model.

Examples

1 Get mask as an object using a masked block's path.
   \[
p = \text{Simulink.Mask.get('myModel/Subsystem')};
\]

2 Get the number of parameters in the mask.
   \[
p.numParameters;
\]
See Also

“Create Block Masks” | Simulink.Mask
removeDialogControl

Class: Simulink.Mask
Package: Simulink

Remove dialog control element from mask dialog box

Syntax

successIndicator = maskVariable.removeDialogControl(controlIdentifier)

Description

successIndicator = maskVariable.removeDialogControl(controlIdentifier) removes dialog control element, specified by controlIdentifier, like text, hyperlinks, or tabs from a mask dialog box. First get the mask object and assign it to the variable maskVariable.

Successful removal of a dialog control element returns a Boolean value of 1.

Input Arguments

ccontrolIdentifier — Unique identifier for the element
character vector | string scalar

Programmatic identifier for the dialog control element of mask dialog box.

Examples

Remove Dialog Control Element from Mask Dialog Box

% Get mask object on the Gain block in the model Engine.
maskObj = Simulink.Mask.get('Engine/Gain');

% Remove element named AllTab from mask dialog box.

p = maskObj.removeDialogControl('AllTab');

**See Also**

“Create Block Masks” | Simulink.Mask

**Introduced in R2013b**
removeParameter

**Class:** Simulink.Mask  
**Package:** Simulink

Remove parameter from mask dialog box

**Syntax**

```matlab
successIndicator = maskVariable.removeParameter(controlIdentifier)
```

**Description**

successIndicator = maskVariable.removeParameter(controlIdentifier)
removes parameter, specified by controlIdentifier, like edit, check box, popup from an existing mask dialog box. First get the mask object and assign it to the variable maskVariable.

Successful removal of a parameter returns a Boolean value of 1.

**Input Arguments**

- **controlIdentifier** — Unique identifier for the parameter  
  character vector | string scalar

Programmatic identifier for the parameter of mask dialog box, specified as a character vector.

**Examples**

**Remove Parameter from Mask Dialog Box**

```matlab
% Get mask object on the Gain block in the model Engine.
```
maskObj = Simulink.Mask.get('Engine/Gain');

% Remove parameter named checkbox1 from mask dialog box.

p = maskObj.removeParameter('checkbox1');

**Note** You can also use the index number as the controlIdentifier.

**See Also**
“Create Block Masks” | Simulink.Mask

**Introduced in R2012b**
removeAllParameters

Class: Simulink.Mask
Package: Simulink

Remove all existing parameters from a mask

Syntax

\[ p = \text{Simulink.Mask.get}(\text{blockName}) \]

\[ p . \text{removeAllParameters} \]

Description

\[ p = \text{Simulink.Mask.get}(\text{blockName}) \] gets the mask on the block specified by blockName as a mask object.

\[ p . \text{removeAllParameters} \] deletes all existing parameters from the mask.

Input Arguments

blockName

The handle to the block or the path to the block inside the model.

Examples

1. Get mask as an object using a masked block's path.

\[ p = \text{Simulink.Mask.get}('\text{myModel/Subsystem}') \]

2. Delete all existing parameters from the mask.

\[ p . \text{removeAllParameters} \]
See Also

“Create Block Masks” | Simulink.Mask
set

Class: Simulink.Mask
Package: Simulink

Set the properties of an existing mask

Syntax

p = Simulink.Mask.get(blockName)
p.set(Name,Value)

Description

p = Simulink.Mask.get(blockName) gets the mask on the block specified by blockName as a mask object.

p.set(Name,Value) sets mask properties that you specify using name-value pairs as arguments.

Input Arguments

blockName

The handle to the block or the path to the block inside the model.

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name, Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside quotes. You can specify several name and value pair arguments in any order as Name1, Value1, ..., NameN, ValueN.
**Type**

Text used as title for mask documentation that user sees on clicking Help in the Mask Parameters dialog box.

**Default:** empty

**Description**

Text used as summary for mask documentation that user sees on clicking Help in the Mask Parameters dialog box.

**Default:** empty

**Help**

Text used as body text for mask documentation that user sees on clicking Help in the Mask Parameters dialog box.

**Default:** empty

**Initialization**

MATLAB code that initializes the mask.

**Default:** empty

**SelfModifiable**

Option to set whether the mask can modify itself during simulation.

**Default:** 'off'

**Display**

MATLAB code that draws the mask icon.

**Default:** empty

**IconFrame**

Option to specify whether the mask icon appears inside a visible block frame.

**Default:** 'on'
**MaskIconOpaque**

Option to set the mask icon as opaque or transparent.

**Default:** 'opaque'

**RunInitForIconRedraw**

Option to specify whether Simulink should run mask initialization before executing the mask icon commands.

**Default:** 'off'

**IconRotate**

Option to specify icon rotation.

**Default:** 'none'

**PortRotate**

Option to specify port rotation.

**Default:** 'default'

**IconUnits**

Option to specify whether mask icon is autoscaled, normalized, or scaled in pixels.

**Default:** 'autoscale'

**Examples**

1. Get mask as an object using a masked block’s path.
   
   ```
   p = Simulink.Mask.get('myModel/Subsystem');
   ```

2. Modify the mask so that its mask icon is transparent and its documentation summarizes what it does.
   
   ```
   p.set('IconOpaque','off','Type','Random number generator','Description',...
   'This block generates random numbers.');
   ```
See Also

“Create Block Masks” | Simulink.Mask
addParameterConstraint

Class: Simulink.Mask
Package: Simulink

Add parameter constraint to a mask

Syntax

paramConstraint = maskObj.addParameterConstraint(Name,Value)

Description

paramConstraint = maskObj.addParameterConstraint(Name,Value) adds a constraint to the specified mask. Constraints can only be associated to the Edit type mask parameters.

Input Arguments

maskObj — Block mask handle
mask object

Block mask handle, specified as a mask object. You can use the Simulink.Mask.get command to get the block mask handle. For more information, see Simulink.Mask.get

Data Types: char | cell

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. Name is the argument name and Value is the corresponding value. You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

Name — Mask constraint name
character vector | string
Required field. Must be a valid MATLAB name. Specifies a name for the mask parameter constraint.

**Parameters — Mask parameter name**
cell array of character vector | string

Optional field. Specifies the **Edit** mask parameter name to which you want to associate the constraint.

**Rules — Rule for mask parameter constraint**
DataType | Dimension | Complexity | Sign | Finiteness | Minimum | Maximum | CustomConstraint

Required field. Rules are defined within curly braces. A constraint can have single or multiple rules.

**Name-Value Pairs for Rules**

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>DataType</strong></td>
<td>double, single, numeric, integer, int8, uint8, int16, uint16, int32, uint32, boolean, enum, fixdt</td>
</tr>
<tr>
<td><strong>Dimension</strong></td>
<td>scalar, rowvector, columnvector, 2dmatrix, ndmatrix</td>
</tr>
<tr>
<td><strong>Complexity</strong></td>
<td>real, complex</td>
</tr>
<tr>
<td><strong>Sign</strong></td>
<td>positive, negative, zero</td>
</tr>
<tr>
<td><strong>Finiteness</strong></td>
<td>finite, inf, -inf, NaN</td>
</tr>
<tr>
<td><strong>Minimum</strong></td>
<td>character vector</td>
</tr>
<tr>
<td><strong>Maximum</strong></td>
<td>character vector</td>
</tr>
<tr>
<td><strong>CustomConstraint</strong></td>
<td>Valid MATLAB expression returning logical true or false.</td>
</tr>
</tbody>
</table>

**Output Arguments**

**paramConstraint — Mask parameter constraint**
character vector | string
Handle to the mask parameter constraint, returned as a character vector. You can associate a constraint to the mask parameter either during or after creating the constraint.

**Examples**

**Create Mask Constraint with Single Rule**

```matlab
% Get block mask handle.
maskObj = Simulink.Mask.get(gcb);

% Create mask constraint with single rule.
paramConstraint = maskObj.addParameterConstraint('Name','const2',... 'Parameters','Parameter2', 'Rules', {'DataType', 'uint8'})

ans =

Constraints with properties:

    Name: 'const2'
    ConstraintRules: [1x1 Simulink.Mask.ParameterConstraintRules]
```

**Create Mask Constraint with Multiple Rules**

```matlab
% Get block mask handle.
maskObj = Simulink.Mask.get(gcb);

% Create mask constraint with multiple rules.
paramConstraint = maskObj.addParameterConstraint('Name','const3',... 'Parameters','Parameter3', 'Rules', {{'DataType', 'uint8'}, {'DataType', {'fixdt(1,8,4)'}}})

ans =

Constraints with properties:

    Name: 'const3'
    ConstraintRules: [1x2 Simulink.Mask.ParameterConstraintRules]
```
See Also

“Create Block Masks” | Simulink.Mask

Introduced in R2018a
removeParameterConstraint

Class: Simulink.Mask
Package: Simulink

Delete a mask parameter constraint

Syntax

maskObj.removeParameterConstraint(paramConstraint)

Description

maskObj.removeParameterConstraint(paramConstraint) deletes the specified mask parameter constraint.

Input Arguments

maskObj — Block mask handle

mask object

Handle to the block mask, specified as mask object. You can use the Simulink.Mask.get command to get the block mask handle. For more information, see Simulink.Mask.get.

Data Types: char | cell

paramConstraint — Mask constraint name

character vector | string

Name of the mask parameter constraint to be deleted, specified as character vector.

Examples

% Get block mask handle.
maskObj = Simulink.Mask.get(gca);
removeParameterConstraint

% Remove mask constraint with name 'const1'.
maskObj.removeParameterConstraint('const1')

See Also
“Create Block Masks” | Simulink.Mask

Introduced in R2018a
removeCrossParameterConstraint

Class: Simulink.Mask
Package: Simulink

Delete a cross-parameter constraint

Syntax

maskObj.removeCrossParameterConstraint(CrossConstraint)

Description

maskObj.removeCrossParameterConstraint(CrossConstraint) deletes the specified cross-constraint.

Input Arguments

maskObj — Block mask handle

mask object

Handle to the block mask, specified as mask object. You can use the Simulink.Mask.get command to get the block mask handle. For more information, see Simulink.Mask.get.

Data Types: char | cell

CrossConstraint — Cross constraint name

character vector | string

Name of the cross-constraint to be removed, specified as character vector.

Examples

% Get block mask handle.
maskObj = Simulink.Mask.get(gcb);
% Remove cross-constraint of name 'const1'.
maskObj.removeCrossParameterConstraint('const1')

See Also

“Create Block Masks” | Simulink.Mask

Introduced in R2018a
**removeAllParameterConstraints**

**Class:** Simulink.Mask  
**Package:** Simulink

Delete all mask parameter constraints

**Syntax**

```matlab
maskObj.removeAllParameterConstraints()
```

**Description**

`maskObj.removeAllParameterConstraints()` deletes all the parameter constraints from a mask.

**Input Arguments**

`maskObj` — Block mask handle
mask object

Handle to the block mask. You can use the `Simulink.Mask.get` command to get the block mask handle. For more information see, `Simulink.Mask.get`

**Examples**

```matlab
% Get block mask handle.
maskObj = Simulink.Mask.get(gcb);

% Remove all constraints from the mask.
maskObj.removeAllParameterConstraints()
```
See Also
“Create Block Masks” | Simulink.Mask

Introduced in R2018a
removeAllCrossParameterConstraints

Class: Simulink.Mask
Package: Simulink

Delete all cross-parameter constraints from a mask

Syntax

maskObj.removeAllCrossParameterConstraints()

Description

maskObj.removeAllCrossParameterConstraints() deletes all cross-constraints from a mask.

Examples

% Get block mask handle.
maskObj = Simulink.Mask.get(gcb);

% Remove all cross constraints from the mask.
maskObj.removeAllCrossParameterConstraints()

See Also

“Create Block Masks” | Simulink.Mask

Introduced in R2018a
getParameterConstraint

Class: Simulink.Mask
Package: Simulink

Get mask parameter constraint properties

Syntax

```
paramConstraint = maskObj.getParameterConstraint(paramConstraintName)
```

Description

```
paramConstraint = maskObj.getParameterConstraint(paramConstraintName)
```

gets the properties of a mask parameter constraint.

Input Arguments

- **maskObj** — Block mask handle
  
  mask object
  
  Handle to the block mask, specified as mask object. You can use the Simulink.Mask.get command to get the block mask handle. For more information, see Simulink.Mask.get.
  
  Data Types: char | cell

- **paramConstraintName** — Mask constraint name
  
  character vector | string
  
  Name of the constraint of which you want get the properties, specified as character vector.
**Output Arguments**

`paramConstraint — Mask constraint property

`cell array of character vector

Constraint properties, returned as a cell array.

**Examples**

```matlab
% Get block mask handle.
maskObj = Simulink.Mask.get(gca);

% Find parameters associated with the constraint.
paramConstraint = maskObj.getParameterConstraint('const3')

ans =

    Constraints with properties:

        Name: 'const3'
    ConstraintRules: [1x2 Simulink.Mask.ParameterConstraintRules]
```

**See Also**

“Create Block Masks” | `Simulink.Mask`

**Introduced in R2018a**
getCrossParameterConstraint

Class: Simulink.Mask
Package: Simulink

Get cross-parameter constraint

Syntax

CrossConstraint = maskObj.getCrossParameterConstraint(CrossConstraintName)

Description

CrossConstraint = maskObj.getCrossParameterConstraint(CrossConstraintName) gets the properties of a cross parameter constraint on a mask. Apply a cross-parameter constraint to specify rules between mask parameter values.

Input Arguments

maskObj — Block mask handle
mask object

Handle to the block mask, specified as mask object. You can use the Simulink.Mask.get command to get the block mask handle. For more information, see Simulink.Mask.get.

Data Types: char | cell

CrossConstraintName — Cross-constraint name
character vector | string

Name of the cross-parameter constraint for which you get the constraint properties, specified as the mask object.
Output Arguments

CrossConstraint — Cross-constraint property
cell array

Cross-parameter constraint properties, returned as a cell array.

Examples

% Get block mask handle.
maskObj = Simulink.Mask.get(gcb);

% Get cross constraint.
CrossConstraint = maskObj.getCrossParameterConstraint('crossparam1')

ans =

CrossParameterConstraints with properties:
    Name: 'crossparam1'
    Rule: 'Parameter2 > Parameter3'
  ErrorMessage: ''

See Also

“Create Block Masks” | Simulink.Mask

Introduced in R2018a
getAssociatedParametersOfConstraint

Class: Simulink.Mask
Package: Simulink

Get mask parameters associated with a constraint

Syntax

maskParam = maskObj.getAssociatedParametersOfConstraint(paramConstraintName)

Description

maskParam = maskObj.getAssociatedParametersOfConstraint(paramConstraintName) gets the parameters that are associated with a mask constraint.

Input Arguments

maskObj — Block mask handle
mask object

Handle to the block mask, specified as mask object. You can use the Simulink.Mask.get command to get the block mask handle. For more information, see Simulink.Mask.get.

Data Types: char | cell

paramConstraintName — Mask constraint name
character vector | string

Name of the constraint for which you want to find the associated mask parameters, specified as character vector.
Output Arguments

`maskParam — Mask parameter name
cell array of character vector`

Mask parameter, specified as a cell array.

Examples

```matlab
% Get block mask handle.
maskObj = Simulink.Mask.get(gca);

% Find parameters associated with the constraint.
maskParam = maskObj.getAssociatedParametersOfConstraint('const3')

ans =

1×1 cell array

{'Parameter1'}
```

See Also

“Create Block Masks” | `Simulink.Mask`

Introduced in R2018a
addCrossParameterConstraint

Class: Simulink.Mask
Package: Simulink

Add cross-parameter constraint

Syntax

CrossConstraint = maskObj.addCrossParameterConstraint(Name,Value)

Description

CrossConstraint = maskObj.addCrossParameterConstraint(Name,Value)
adds a constraint among parameters of a mask.

Input Arguments

maskObj — Block mask handle
mask object

Block mask handle, specified as a mask object. You can use the Simulink.Mask.get command to get the block mask handle. For more information, see Simulink.Mask.get.

Data Types: char | cell

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. Name is the property name and Value is the corresponding value. You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

Name — Cross-parameter constraint name
character vector | string
Cross-parameter constraint name, specified as a character vector. Must be a valid MATLAB value.

**Rule — Cross-parameter constraint rule**
MATLAB expression | string

Cross-parameter constraint rule, specified as a valid MATLAB expression that returns logical true or false. You can specify multiple rules by separating them with a logical operators like &&. For example, parameter1 > parameter2 && parameter2 > parameter3. Here, parameter1, parameter2 and parameter3 are parameters of a mask.

**ErrorMessage — Error message**
character vector | string

Optional field. Specifies the error message to be displayed when the cross parameter constraint rule is not satisfied. You can specify the error message as character vector or as a message catalog ID. If you use the message catalog ID to specify an error, the error message must not have any holes in it. Simulink displays a default error message if no user-defined error message is found.

**Output Arguments**

**CrossConstraint — Cross parameter constraint**
cell array

Handle to the cross-parameter constraint, returned as a cell array.

**Examples**

```matlab
% Get block mask handle.
maskObj = Simulink.Mask.get(gcb);

% Add cross parameter constraint.
CrossConstraint = maskObj.addCrossParameterConstraint('Name','crossconstraint1',...
'Rule','upperbound > lowerbound','ErrorMessage','Incorrect value specified.')
```

`ans =`
CrossParameterConstraints with properties:

Name: 'crossconstraint1'
Rule: 'upperbound > lowerbound'
ErrorMessage: 'Incorrect value specified.'

See Also
“Create Block Masks” | Simulink.Mask

Introduced in R2018a
Simulink.Mask.Constraints class

Package: Simulink.Mask

Create Mask Constraint

Description

Use an instance of the Simulink.Mask.Constraint to add or remove a parameter constraint rule.

Properties

DataType

Specifies the data type associated with the constraint rule.

Type: double, single, numeric, integer, int8, uint8, int16, uint16, int32, uint32, boolean, enum, fixdt, string

Default: Empty

Dimension

Specifies the dimension associated with the constraint rule.

Type: scalar, rowvector, columnvector, 2dmatrix, ndmatrix

Default: Empty

Complexity

Specifies the complexity associated with the constraint rule.

Type: real, complex

Default: Empty
Sign
Specifies the sign associated with the constraint rule.

Type: positive, negative, zero

Default: Empty

Finiteness
Specifies the finiteness associated with the constraint rule.

Type: finite, inf, -inf, NaN

Default: Empty

Range
Specifies the range associated with the constraint rule.

Type: Minimum, Maximum

Default: Empty

CustomConstraint
Specifies the custom constraint associated with the constraint rule.

Type: Valid MATLAB expression

Default: Empty

CustomErrorMessage
Specifies the custom error message associated with the custom constraint rule.

Type: Character vector

Default: Empty
Methods

addParameterConstraintRule  Add rules to a parameter constraint
removeParameterConstraintRule  Delete a mask parameter constraint rule

See Also

Introduced in R2018a
addParameterConstraintRule

Class: Simulink.Mask.Constraints
Package: Simulink.Mask

Add rules to a parameter constraint

Syntax

paramConstRule = paramConstraint.addParameterConstraintRule(Name,Value)

Description

paramConstRule = paramConstraint.addParameterConstraintRule(Name,Value) adds rule to a parameter constraint.

Input Arguments

paramConstraint — Handle to mask constraint
  constraint object

Handle to the mask parameter constraint for which you want to add constraint rules, specified as constraint object.

Data Types: char | cell

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. Name is the argument name and Value is the corresponding value. You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.
**Name-Value Pair for Rules**

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>DataType</strong></td>
<td>double, single, numeric, integer, int8, uint8, int16, uint16, int32, uint32, boolean, enum, fixdt</td>
</tr>
<tr>
<td><strong>Dimension</strong></td>
<td>scalar, rowvector, columnvector, 2dmatrix, ndmatrix</td>
</tr>
<tr>
<td><strong>Complexity</strong></td>
<td>real, complex</td>
</tr>
<tr>
<td><strong>Sign</strong></td>
<td>positive, negative, zero</td>
</tr>
<tr>
<td><strong>Finiteness</strong></td>
<td>finite, inf, -inf, NaN</td>
</tr>
<tr>
<td><strong>Minimum</strong></td>
<td>string</td>
</tr>
<tr>
<td><strong>Maximum</strong></td>
<td>string</td>
</tr>
<tr>
<td><strong>CustomConstraint</strong></td>
<td>Valid MATLAB expression</td>
</tr>
</tbody>
</table>

**Output Arguments**

`paramConstRule — Mask constraint rule`  
cell array

Mask constraint rule, specified as a cell array.

**Examples**

```matlab
% Get mask constraint handle
paramConstraint = maskObj.getParameterConstraint('const3');

% Add rules to the constraint.
paramConstRule = paramConstraint.addParameterConstraintRule('DataType','int8')
```

```
ans =
    ParameterConstraintRules with properties:
        DataType: 'int8'
```
See Also
“Create Block Masks” | Simulink.Mask

Introduced in R2018a
removeParameterConstraintRule

Class: Simulink.Mask.Constraints
Package: Simulink.Mask

Delete a mask parameter constraint rule

Syntax

paramConstraint.removeParameterConstraintRule(RuleIndex)

Description

paramConstraint.removeParameterConstraintRule(RuleIndex) deletes the specified constraint rule from a mask parameter constraint.

Input Arguments

paramConstraint — Handle to constraint
constraint object

Handle to mask parameter constraint of which you want to remove constraint rule, specified as an object.

Data Types: char | cell

RuleIndex — Constraint rule index
integer

Index number of the mask constraint rule, specified as an integer.

Examples

% Get block mask handle.
maskObj = Simulink.Mask.get(gcb);
% Get mask constraint handle.
paramConstraint = maskObj.getParameterConstraint('const3');

% Remove mask constraint rule.
paramConstraint.removeParameterConstraintRule(1)

See Also
“Create Block Masks” | Simulink.Mask

Introduced in R2018a
**addColumn**

**Class:** Simulink.dialog.parameter.CustomTable  
**Package:** Simulink.dialog.parameter

Add new column in the custom table

**Syntax**

```
tableControl.addColumn(Name, Value)
```

**Description**

```
tableControl.addColumn(Name, Value) adds a column with the specified properties in the custom table.
```

**Input Arguments**

```
tableControl — Custom table handle  
  
  table object
```

Handle to the custom table, specified as an object. You can use the `getDialogControl` command to get the custom table handle. For more information, see `getDialogControl`.

Consider, for example, `tableControl = maskObj.getDialogControl(TableName)`. Here, `maskObj` is the mask object and `TableName` is the name of the custom table.

**Data Types:** char vector | cell array

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.
Examples

% Get block mask handle.
maskObj = Simulink.Mask.get(gca);

% Get custom table handle.
tableControl = maskObj.getDialogControl('myTable');

% Add column.
tableControl.addColumn( 'Name', 'HDL Name', 'Type', 'edit' );

See Also

“Create Block Masks” | Simulink.Mask

Introduced in R2019a
**setTableCell**

*Class:* Simulink.dialog.parameter.CustomTable  
*Package:* Simulink.dialog.parameter

Set the properties of a custom table cell

**Syntax**

```matlab
tableCell = tableControl.setTableCell([rowIdx colIdx], Type customtable, Evaluate, off)
```

**Description**

```matlab
tableCell = tableControl.setTableCell([rowIdx colIdx], Type customtable, Evaluate, off)
```
sets the specified properties of a cell in a custom table.

**Input Arguments**

- **tableControl** — Custom table handle  
  table object

Handle to the custom table, specified as an object. You can use the `getDialogControl` command to get the custom table handle. For more information, see `getDialogControl`.

Consider, for example, `tableControl = maskObj.getDialogControl(TableName)`. Here, `maskObj` is the mask object and `TableName` is the name of the custom table.

Data Types: Numeric Array Name Value Pair

- **rowIdx** — Row index

Row index number of the custom table cell for which the value is being fetched.

For example, '3'.
colIdx — Column index

Column index number of the custom table cell for which the value is being fetched.

For example, '5'.

Examples

% Get block mask handle.
maskObj = Simulink.Mask.get(gcb);

% Get custom table handle.
tableControl = maskObj.getDialogControl('myTable');

% Get cell properties.
tableCell = tableControl.setTableCell([2 2]);

See Also

“Create Block Masks“ | Simulink.Mask

Introduced in R2019a
getNumberOfColumns

**Class:** Simulink.dialog.parameter.CustomTable  
**Package:** Simulink.dialog.parameter

Get value of a table cell

**Syntax**

```matlab
tableControl().getNumberOfColumns()
```

**Description**

`numberOfColumns = tableControl()` gets the number of columns in a custom table.

**Output Arguments**

- `numberOfColumns` — Table cell value  
  `integer`

  Number of columns in the custom table returned as an integer.

**Examples**

```matlab
% Get block mask handle.  
maskObj = Simulink.Mask.get(gca);

% Get custom table handle.  
tableControl = maskObj.getDialogControl('myTable');

% Get cell value.  
numberOfColumns = tableControl.getNumberOfColumns();
```
See Also
“Create Block Masks” | Simulink.Mask

Introduced in R2019a
getNumberOfRows

**Class:** Simulink.dialog.parameter.CustomTable  
**Package:** Simulink.dialog.parameter

Get the number of rows available in a custom table

**Syntax**

```plaintext
numberOfRows = tableControl()
```

**Description**

`numberOfRows = tableControl()` gets the number of rows in a custom table.

**Output Arguments**

`numberOfRows` — Number of Rows  
integer

Number of rows in the custom table returned as an integer.

**Examples**

```plaintext
% Get block mask handle.
maskObj = Simulink.Mask.get(gca);

% Get custom table handle.
tableControl = maskObj.getDialogControl('myTable');

% Get number of rows.
numberOfRows = tableControl.getNumberOfRows();
```
See Also
“Create Block Masks” | Simulink.Mask

Introduced in R2019a
addRow

Class: Simulink.dialog.parameter.CustomTable
Package: Simulink.dialog.parameter

Add a new row in the custom table

Syntax

NewRow = tableControl.addRow(RowValue)

Description

NewRow = tableControl.addRow(RowValue) adds a row with the specified properties in the custom table.

Input Arguments

tableControl — Custom table handle
table object

Handle to the custom table, specified as an object. You can use the getDialogControl command to get the custom table handle. For more information, see getDialogControl.

Consider, for example, tableControl = maskObj.getDialogControl(TableName). Here, maskObj is the mask object and TableName is the name of the custom table.

Data Types: char vector

RowValue — Row Values for each column
table object

Row values specified for the number of columns available. If five values are provided for RowValue in a custom table with five columns, one row is added to the table. If 10 row values are added for such a table, then two rows are added.
Consider, for example, `tableControl = maskObj.getDialogControl(TableName)`. Here, `maskObj` is the mask object and `TableName` is the name of the custom table.

Data Types: `char vector | cell array`

### Examples

```matlab
% Get block mask handle.
maskObj = Simulink.Mask.get(gcb);

% Get custom table handle.
tableControl = maskObj.getDialogControl('myTable');

% Add a row with six column values.
tableControl.addRow( 'sig5', 'Input', 'Inherit', 'Inherit', 'on', 'Inherit' );
```

### See Also

“Create Block Masks” | Simulink.Mask

**Introduced in R2019a**
getChangedCells

Class: Simulink.dialog.parameter.CustomTable
Package: Simulink.dialog.parameter

Get all the changed cells of a custom table

Syntax

changedCells = tableControl.getChangedCells()

Description

changedCells = tableControl.getChangedCells() gets the properties of a cell in the custom table.

Output Arguments

changedCells — Cells in a custom table whose values have been changed
cell array

Shows the changed cells in a custom table.

Examples

% Get block mask handle.
maskObj = Simulink.Mask.get(gca);

% Get custom table handle.
tableControl = maskObj.getDialogControl('myTable');

% Get changed cells.
changedCell = tableControl.getchangedCells();
ans =

1x4 cell array

{1x2 double}    {1x2 double}    {1x2 double}    {1x2 double}

Output: Cell Array
Input: None

See Also
“Create Block Masks” | Simulink.Mask

Introduced in R2019a
getColumn

Class: Simulink.dialog.parameter.CustomTable
Package: Simulink.dialog.parameter

Get the column properties

Syntax

FetchColumn = tableControl.getColumn(column)

Description

FetchColumn = tableControl.getColumn(column) fetches the information from the specified column of the custom table.

Input Arguments

tableControl — Custom table handle
table object

Handle to the custom table, specified as an object. You can use the getDialogControl command to get the custom table handle. For more information, see getDialogControl.

Consider, for example, tableControl = maskObj.getDialogControl(TableName). Here, maskObj is the mask object and TableName is the name of the custom table.

Data Types: integer

columnIndex — Column Index
string | character array

Index of the column in the custom table.

For example, '4'.

5-1142
**Output Arguments**

`fetchcolumn — Column properties`  
column object  
Column object

**Examples**

```matlab
% Get block mask handle.
maskObj = Simulink.Mask.get(gca);

% Get custom table handle.
tableControl = maskObj.getDialogControl('myTable');

% Get properties of column 3.
tableControl.getColumn(3);

ans =
    TableParamColumnInfo with properties:
        Name: 'Data Type'
        Type: 'popup'
       Enabled: 'on'
   TypeOptions: {4×1 cell}
```

**See Also**

“Create Block Masks” | `Simulink.Mask`

**Introduced in R2019a**
getSelectedRows

Class: Simulink.dialog.parameter.CustomTable
Package: Simulink.dialog.parameter

Get all the selected rows of a custom table

Syntax

**selectedRows = tableControl.getSelectedRows()**

Description

**selectedRows = tableControl.getSelectedRows()** fetches the information from the specified column of a custom table.

Output Arguments

**selectedRows — Row information**
numeric array

Value of the specified rows, returned as a numeric array.

Examples

```matlab
% Get block mask handle.
maskObj = Simulink.Mask.get(gcb);

% Get custom table handle.
tableControl = maskObj.getDialogControl('myTable');

% Get selected rows.
selectedRows = tableControl.getSelectedRows();
```
ans =

3   4

See Also
“Create Block Masks” | Simulink.Mask

Introduced in R2019a
**getTableCell**

**Class:** Simulink.dialog.parameter.CustomTable  
**Package:** Simulink.dialog.parameter

Get the properties of a custom table cell

**Syntax**

```plaintext
tableCell = tableControl.getTableCell([rowIdx colIdx])
```

**Description**

`tableCell = tableControl.getTableCell([rowIdx colIdx])` gets the properties of a cell in a custom table.

**Input Arguments**

- **tableControl** — Custom table handle  
  Table object  
  Handle to a custom table, specified as an object. You can use the `getDialogControl` command to get the custom table handle. For more information, see `getDialogControl`.  
  Consider, for example, `tableControl = maskObj.getDialogControl(TableName)`. Here, `maskObj` is the mask object and `TableName` is the name of the custom table.

  Data Types: numeric array  

- **rowIdx** — Row index  
  Row index number of the custom table cell for which the value is being fetched.  
  For example, '3'.

- **colIdx** — Column index  
  Column index number of the custom table cell for which the value is being fetched.
Column index number of the custom table cell for which the value is being fetched.
For example, '5'.

**Output Arguments**

`tableCell — Table cell object
`cell object

Properties of the table cell, returned as a cell array.

**Examples**

```matlab
% Get block mask handle.
maskObj = Simulink.Mask.get(gca);

% Get custom table handle.
tableControl = maskObj.getDialogControl('myTable');

% Get cell properties.
tableCell = tableControl.getTableCell([2 2]);

tableCell =

    CustomTableParamCellObject with properties:

        Value: 'Inherit'
        Type: 'popup'
    Enabled: 'off'
        TypeOptions: {4×1 cell}

    tableCell.Value = 'Value'
```

**See Also**

“Create Block Masks” | Simulink.Mask

**Introduced in R2019a**
**getValue**

**Class:** Simulink.dialog.parameter.CustomTable  
**Package:** Simulink.dialog.parameter

Get value of a table cell

**Syntax**

```matlab
tableCellValue = tableControl.getValue([rowIdx colIdx])
```

**Description**

`tableCellValue = tableControl.getValue([rowIdx colIdx])` gets the value of a cell in a custom table.

**Input Arguments**

- `tableControl` — Custom table handle  
  `table object`  
  Handle to the custom table, specified as an object. You can use the `getDialogControl` command to get the custom table handle. For more information, see `getDialogControl`.

  Consider, for example, `tableControl = maskObj.getDialogControl(TableName)`. Here, `maskObj` is the mask object and `TableName` is the name of the custom table.

  Data Types: numeric array

- `rowIdx` — Row index  
  Row index number of the custom table cell for which the value is being fetched.

  For example, `[2 2]`. 
colIdx — Column index

Column index number of the custom table cell for which the value is being fetched.

For example, [2 2].

Output Arguments

tableCellValue — Table cell value
character vector

Value of the table cell, returned as a character vector.

Examples

% Get block mask handle.
maskObj = Simulink.Mask.get(gcb);

% Get custom table handle.
tableControl = maskObj.getDialogControl('myTable');

% Get cell value.
tableCellValue = tableControl.getValue([2 2]);

ans =
   'Input'

See Also

“Create Block Masks” | Simulink.Mask

Introduced in R2019a
insertColumn

**Class:** Simulink.dialog.parameter.CustomTable  
**Package:** Simulink.dialog.parameter

Insert a column at a specific position in the custom table

**Syntax**

```matlab
tableControl.insertColumn(columnIndex[Name, Value])
```

**Description**

`tableControl.insertColumn(columnIndex[Name, Value])` inserts a column with the specified properties at a specified index in a custom table.

**Input Arguments**

- `tableControl` — Custom table handle  
  `table object`

Handle to the custom table, specified as an object. You can use the `getDialogControl` command to get the custom table handle. For more information, see `getDialogControl`.

Consider, for example, `tableControl = maskObj.getDialogControl(TableName)`. Here, `maskObj` is the mask object and `TableName` is the name of the custom table.

- `columnIndex` — Column Index  
  `string`

Data Types: `char` vector | cell array

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`. 

- `columnIndex` — Column Index  
  `string`
Column index number of the custom table cell at which the new column is to be inserted. For example, '4'.

**Name**

Name of the column.

**Default:** generated name

**Type**

Type of control that is used to specify the value of this parameter. The permitted values are: edit, checkbox, popup.

**Default:** edit

**TypeOptions**

Options that are displayed within a popup control or in a promoted parameter. This field is a cell array.

**Default:** empty

**Enabled**

Option to specify whether the user can set parameter value.

**Default:** 'on'

**Examples**

```matlab
% Get block mask handle.
maskObj = Simulink.Mask.get(gca);

% Get custom table handle.
tableControl = maskObj.getDialogControl('myTable');

% Insert a column at the fifth position.
tableControl.insertColumn(5,'Name', 'HDL Name', 'Type', 'edit');
```
See Also
“Create Block Masks” | Simulink.Mask

Introduced in R2019a
insertRow

Class: Simulink.dialog.parameter.CustomTable
Package: Simulink.dialog.parameter

Insert a row at a specific position in a custom table

Syntax

InsertedRow = tableControl.insertRow(rowIndex,string1,string2)

Description

InsertedRow = tableControl.insertRow(rowIndex,string1,string2) inserts a row with specified properties at the specified index in a custom table.

Input Arguments

tableControl — Custom table handle
table object

Handle to the custom table, specified as an object. You can use the getDialogControl command to get the custom table handle. For more information, see getDialogControl.

Consider, for example, tableControl = maskObj.getDialogControl(TableName). Here, maskObj is the mask object and TableName is the name of the custom table.

Data Types: char vector | cell array

rowIndex — Row Index

Index in table at which the new row is to be inserted.

For example, '4'.
Examples

% Get block mask handle.
maskObj = Simulink.Mask.get(gcb);

% Get custom table handle.
tableControl = maskObj.getDialogControl('myTable');

% Insert Row.
tableControl.insertRow( rowIndex,'insertSig4', 'Input', 'Inherit', 'Inherit', 'on', 'Inherit' );

See Also
“Create Block Masks” | Simulink.Mask

Introduced in R2019a
removeColumn

Class: Simulink.dialog.parameter.CustomTable
Package: Simulink.dialog.parameter

Remove an existing column from a specific position in a custom table

Syntax

tableControl.removeColumn(columnIndex)

Description

tableControl.removeColumn(columnIndex) removes a column from the specified index in a custom table.

Input Arguments

**tableControl** — Custom table handle

table object

Handle to the custom table, specified as an object. You can use the `getDialogControl` command to get the custom table handle. For more information, see `getDialogControl`.

Consider, for example, `tableControl = maskObj.getDialogControl(TableName)`. Here, `maskObj` is the mask object and `TableName` is the name of the custom table.

Data Types: string

**columnIndex** — Column Index

integer

Column index number of the custom table from which the column is to be removed.

For example, '4'.

5-1155
Examples

% Get block mask handle.
maskObj = Simulink.Mask.get(gcb);

% Get custom table handle.
tableControl = maskObj.getDialogControl('myTable');

% Remove the third column.
tableControl.removeColumn(3);

See Also

“Create Block Masks” | Simulink.Mask

Introduced in R2019a
**removeRow**

**Class:** Simulink.dialog.parameter.CustomTable  
**Package:** Simulink.dialog.parameter

Remove an existing row from a specific position in a custom table

**Syntax**

```matlab
tableControl.removeRow(rowIndex)
```

**Description**

`tableControl.removeRow(rowIndex)` removes the row from a specified position in a custom table.

**Input Arguments**

- `tableControl` — Custom table handle  
  - table object  
  Handle to the custom table, specified as an object. You can use the `getDialogControl` command to get the custom table handle. For more information, see `getDialogControl`.  
  Consider, for example, `tableControl = maskObj.getDialogControl(TableName)`. Here, `maskObj` is the mask object and `TableName` is the name of the custom table.

- `rowIndex` — Row Index  
  Row index number of the custom table from which the row is to be removed.

  For example, '4'.

Data Types: `integer`
Examples

% Get block mask handle.
maskObj = Simulink.Mask.get(gcb);

% Get custom table handle.
tableControl = maskObj.getDialogControl('myTable');

% Remove row with index number 6.
tableControl.removeRow(6);

See Also
“Create Block Masks” | Simulink.Mask

Introduced in R2019a
swapRows

Class: Simulink.dialog.parameter.CustomTable
Package: Simulink.dialog.parameter

Swap two existing rows in a custom table

Syntax

```
tableControl.swapRows(rowIndex1, rowIndex2)
```

Description

tableControl.swapRows(rowIndex1, rowIndex2) swaps specified rows in a custom table.

Input Arguments

- **tableControl** — Custom table handle
  - table object
  
  Handle to the custom table, specified as an object. You can use the `getDialogControl` command to get the custom table handle. For more information, see `getDialogControl`.

  Consider, for example, `tableControl = maskObj.getDialogControl(TableName)`. Here, `maskObj` is the mask object and `TableName` is the name of the custom table.

  Data Types: integer

- **rowIndex1** — Row Index value
  - integer
  
  Row index number of the first row to be swapped with another row.

  For example, '4'.
**rowIndex2 — Row Index value**

integer

Row index number of the second row to be swapped with another row.

For example, '5'.

### Examples

```matlab
% Get block mask handle.
maskObj = Simulink.Mask.get(gcb);

% Get custom table handle.
tableControl = maskObj.getDialogControl('myTable');

% Swap rows 3 and 6.
tableControl.swapRows(3, 6);
```

### See Also

“Create Block Masks” | Simulink.Mask

**Introduced in R2019a**
Simulink.dialog.parameter.CustomTable class

Package: Simulink.dialog.parameter

Create custom tables programmatically

Description

Use an instance of Simulink.dialog.parameter.CustomTable class to perform the following operations:

• Create and remove custom tables
• Insert, remove, and get values of columns.
• Insert, remove, swap, and add rows.
Methods

addColumn Add new column in the custom table
setTableCell Set the properties of a custom table cell
getNumberOfColumns Get value of a table cell
getNumberOfRows Get the number of rows available in a custom table
addRow Add a new row in the custom table
getChangedCells Get all the changed cells of a custom table
column Get the column properties
getSelectedRows Get all the selected rows of a custom table
getTableCell Get the properties of a custom table cell
getValue Get value of a table cell
insertColumn Insert a column at a specific position in the custom table
insertRow Insert a row at a specific position in a custom table
removeColumn Remove an existing column from a specific position in a custom table
removeRow Remove an existing row from a specific position in a custom table
swapRows Swap two existing rows in a custom table

See Also

Topics
“Control Masks Programmatically”
“Create Block Masks”
Simulink.dialog.ListboxControl class

Package: Simulink.dialog

Control listbox programmatically

Description

Use an instance of Simulink.dialog.ListBoxControl class to perform the following operations:

- Get selected items from the listbox
- Set selected items in the listbox

Properties

Name

Uniquely identifies the dialog control and is a required field.

Type: character vector

Prompt

Specifies the text displayed on the button dialog control.

Type: character vector

Default: empty

Type Options

Specifies the options shown in the listbox.

Type: cell array

Default: empty
**Callback**
Specifies the MATLAB command(s) to be executed when the dialog control is invoked.

**Type:** character vector

**Default:** empty

**Row**
Specifies whether dialog control is placed on the current row or on a new row.

**Type:** character vector

**Value:** 'current'|'new'

**Default:** 'current'

**Enabled**
Indicates whether container is active on the mask dialog box.

**Type:** boolean

**Value:** 'on'|'off'

**Default:** 'on'

**Visible**
Indicates whether container is displayed on the mask dialog box.

**Type:** boolean

**Value:** 'on'|'off'

**Default:** 'on'

**Multiselect**
Indicates whether container is enabled for multiple selection.

**Type:** boolean
Value: 'on' | 'off'

Default: 'on'

Methods

setSelectedItems Set values for a listbox
getSelectedItems Get the values of selected items from a listbox

See Also

Topics
“Control Masks Programmatically”
“Create Block Masks”
**setSelectedItems**

**Class:** Simulink.dialog.ListboxControl  
**Package:** Simulink.dialog

Set values for a listbox

**Syntax**

```matlab
setItems = listboxControl.setSelectedItems({string1,'string2',...})
```

**Description**

```matlab
setItems = listboxControl.setSelectedItems({string1,'string2',...})
```

set the values for the selected items in the listbox.

**Input Arguments**

- `listControl`  
  **Lisbox handle**
  table object

Handle to the listbox, specified as an object. You can use the `getDialogControl` command to get the listbox handle. For more information, see `getDialogControl`.

For example, `litboxControl = maskObj.getDialogControl('listboxname')`. Here, `maskObj` is the mask object and `listboxname` is the name of the listbox.

**Data Types:** cell array

- `string1`  
  **value to be set**
  string

Value to be set for selected items in the listbox

For example, 'Hello'.
Examples

% Get block mask handle
maskObj = Simulink.Mask.get(gcb);

% Get listbox handle
listboxControl = maskObj.getDialogControl('Control2');

% Set selected items
listboxControl.setSelectedItems({'Hello','World'});

See Also
“Create Block Masks” | Simulink.Mask

Introduced in R2019a
**getSelectedItems**

**Class:** Simulink.dialog.ListBoxControl  
**Package:** Simulink.dialog

Get the values of selected items from a listbox

**Syntax**

```matlab
getItems = listboxControl.getSelectedItems()
```

**Description**

`getItems = listboxControl.getSelectedItems()` fetches the information of the selected items from the listbox.

**Output Arguments**

- `getItems` — *value of selected items*
  - cell array

Value of the selected items, returned as a cell array.

**Examples**

```matlab
% Get block mask handle
maskObj = Simulink.Mask.get(gcb);

% Get listbox handle
listboxControl = maskObj.getDialogControl('Control2');

% Get selected items
getItems = listboxControl.getSelectedItems();
```
getItems =
    1x2 cell array
    {'Hello', 'World'}

See Also
“Create Block Masks” | Simulink.Mask

Introduced in R2019a
Simulink.MaskParameter class

Package: Simulink

Control mask parameters programmatically

Description

Use an instance of Simulink.MaskParameter to set the properties of mask parameters.

Properties

Type

Specifies the mask parameter type.

Type: character vector

Values: 'edit'|'checkbox'|'popup'|'min'|'max'|'promote'|'combobox'|'radiobutton'|'unitd'|'slider'|'dial'|'spinbox'

Default: 'edit'

TypeOptions

Specifies the option for the parameter if it exists, otherwise, it is empty. Applicable for parameters of type popup, radio, Datatypestr, and promote.

Type: cell array of character vectors

Default: {}
**Type:** character vector

**Default:** Auto generated

**Prompt**

Specifies a character vector that appears as the label associated with the parameter on the mask dialog.

**Type:** character vector

**Default:** Empty character vector

**Value**

Specifies the value of the mask parameter.

**Default:** Depends on the type of the parameter.

**Evaluate**

Indicates if the parameter value is to be evaluated in MATLAB or treated as a character vector when the block is evaluated.

**Type:** boolean

**Values:** 'on' | 'off'

**Default:** 'on'

**Tunable**

Indicates if the parameter value can be changed during simulation.

**Type:** boolean

**Values:** 'on' | 'off'

**Default:** 'on'

**NeverSave**

Indicates if the parameter value gets saved in the model file.
Type: boolean
Values: 'on' | 'off'
Default: 'off'

Hidden
Indicates if the parameter should never show on the mask dialog box.
Type: boolean
Values: 'on' | 'off'
Default: 'off'

ReadOnly
Indicates if the parameter on the mask dialog box is editable or is read-only.
Type: boolean
Values: 'on' | 'off'
Default: 'off'

Enabled
Indicates if the parameter is enabled in the mask dialog box.
Type: boolean
Values: 'on' | 'off'
Default: 'on'

Visible
Indicates if the parameter is visible in the mask dialog box.
Type: boolean
Values: 'on' | 'off'
Default: 'on'

ShowTooltip

Indicates if tool tip is enabled for the mask parameter.

Type: boolean

Values: 'on'|'off'

Default: 'on'

Callback

Specifies the MATLAB code that executes when a user changes the parameter value from the mask dialog box.

Type: character vector

Default: Empty character vector

TabName

Specifies the tab name of the mask dialog box where the parameter is displayed.

Type: character vector

Default: Empty character vector

Alias

Specifies the alternate name for mask parameter.

Type: character vector

Default: Empty character vector

DialogControl

Specifies the layout options of mask dialog.

Type: Class of type Simulink.dialog.parameter.<Typeofparameter>
Values:  
'edit'|'checkbox'|'popup'|'min'|'max'|'promote'|'combobox'|'radiobutton'|'unidt'|'slider'|'dial'|'spinbox'

Default: Edit

ConstraintName

Indicates the constraint associated with a parameter. To associate a constraint programmatically the constraint must be already available. To associate a constraint from a mat file use the format `<matfilename>:<constraintname>`.

Type: character vector

Default: Empty character vector

Methods

set Set properties of mask parameters

See Also

Topics
“Control Masks Programmatically”
“Create Block Masks”
set

**Class:** Simulink.MaskParameter  
**Package:** Simulink

Set properties of mask parameters

**Syntax**

Simulink.MaskParameter.set(Name,Value)

**Description**

Simulink.MaskParameter.set(Name,Value) sets the properties of a mask parameter.

**Input Arguments**

**Name-Value Pair Arguments**

Specify optional comma-separated pairs of Name,Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside quotes. You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

**Type**

Type of control that is used to specify the value of this parameter.

**Default:** edit

**TypeOptions**

The options that are displayed within a popup control or in a promoted parameter. This field is a cell array.
**Default:** empty

**Name**
The name of the mask parameter. This name is assigned to the mask workspace variable created for this parameter.

**Default:** empty

**Prompt**
Text that identifies the parameter on the Mask Parameters dialog.

**Default:** empty

**Value**
The default value of the mask parameter in the Mask Parameters dialog.

**Default:** Type specific; depends on the Type of the parameter

**Evaluate**
Option to specify whether parameter must be evaluated.

**Default:** 'on'

**Tunable**
Option to specify whether parameter is tunable.

**Default:** 'on'

**Enabled**
Option to specify whether user can set parameter value.

**Default:** 'on'

**Visible**
Option to set whether mask parameter is hidden or visible to the user.

**Default:** 'on'
Callback

Container for MATLAB code that executes when user makes a change in the Mask Parameters dialog and clicks Apply.

Default: empty

Examples

1. Get mask as an object using a masked block’s path.
   
   ```matlab
   p = Simulink.Mask.get('myModel/Subsystem');
   ```

2. Get a mask parameter.
   
   ```matlab
   a = p.Parameters(1);
   ```

3. Edit mask parameter so it is of type popup, cannot be evaluated.
   
   ```matlab
   a.set('Type','popup','TypeOptions',{'Red' 'Blue' 'Green'},...
   'Evaluate','off');
   ```

See Also

“Create Block Masks” | Simulink.Mask | Simulink.MaskParameter
Simulink.dialog.Control class

Package: Simulink.dialog

Create instances of dialog control

Description

Use an instance of Simulink.dialog.Control class to create, delete, or search dialog controls.

Properties

Name

Uniquely identifies the dialog control element and is a required field.

Type: character vector

See Also

Simulink.dialog.Button | Simulink.dialog.Hyperlink |
Simulink.dialog.Image | Simulink.dialog.Text |
Simulink.dialog.parameter.Control | Simulink.dialog.Container | “Create Block Masks”
Simulink.dialog.Container class

**Package:** Simulink.dialog

Create instances of a container dialog control

**Description**

Use an instance of Simulink.dialog.Container class to add container type dialog control.

**Properties**

**Name**

Uniquely identifies the container dialog control and is a required filed.

*Type:* character vector

**Enabled**

Indicates whether container is active on the mask dialog box.

*Type:* boolean

*Value:* 'on' | 'off'

*Default:* 'on'

**Visible**

Indicates whether container is displayed on the mask dialog box.

*Type:* boolean

*Value:* 'on' | 'off'

*Default:* 'on'
**AlignPrompts**

Allows you to align the parameters vertically on the mask dialog box.

**Type:** boolean  
**Value:** 'on' | 'off'  
**Default:** 'off'

**DialogControls**

Specifies the child dialog controls contained in the container.

**Type:** Simulink.dialog.Control  
**Default:** Empty array

**Methods**

- addDialogControl: Add dialog control elements to mask dialog box  
- removeDialogControl: Remove dialog control element from mask dialog box  
- getDialogControl: Search for a specific dialog control on the mask

**See Also**

addDialogControl

Class: Simulink.dialog.Container
Package: Simulink.dialog

Add dialog control elements to mask dialog box

Syntax

successIndicator = maskObj.addDialogControl(controlType, controlIdentifier)
successIndicator = maskObj.addDialogControl(Name,Value)

Description

successIndicator = maskObj.addDialogControl(controlType, controlIdentifier) adds dialog control elements like text, hyperlinks, or tabs to mask dialog box. First get the mask object and assign it to the variable maskObj

successIndicator = maskObj.addDialogControl(Name,Value) specifies the Name and Value arguments for an element on the mask dialog box. You can specify multiple Name-Value pairs.

Input Arguments

controlType — Value type of dialog control element
character vector

Type of dialog control element, specified

* 'panel'
* 'group'
* 'tabcontainer'
* 'tab'
• 'collapsiblepanel'
• 'text'
• 'image'
• 'hyperlink'
• 'pushbutton'

controlIdentifier — Unique identifier for the element
character vector

Specifies the programmatic identifier for the element of mask dialog box. Use a name that is unique and does not have space between words. For more information, see “Variable Names” (MATLAB).

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name, Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside single quotes ("'") and is case-sensitive. You can specify several name and value pair arguments in any order as Name1, Value1, ..., NameN, ValueN.

Type

Type of control that is used to specify the value of this dialog control element. Type is a required argument. The permitted values are 'panel', 'group', 'tabcontainer', 'tab', 'collapsiblepanel', 'text', 'image', 'hyperlink', and 'pushbutton'. If the parent dialog control type is 'tabcontainer', the child dialog control must be 'tab'.

Name

The identifier of the dialog control element. Name is a required argument. This field is available for all dialog control types.

Prompt

Text that is displayed in the dialog control element on the Mask dialog box. This field is available for all except for panel and image dialog control types.

Default: empty
Enabled

Option to specify whether you can set value for the dialog control element. This field is available for all dialog control types.

**Default:** 'on'

Visible

Option to set whether the dialog control element is hidden or visible to the user. This field is available for all dialog control types.

**Default:** 'on'

Callback

Container for MATLAB code that executes when you edit the dialog control element and click **Apply**. This field is available only for the hyperlink and pushbutton dialog control types.

**Default:** empty

Row

Option to set whether the dialog control is placed in the new row or the same row. This field is available for all dialog control types.

**Default:** empty

FilePath

Contains the path to an image file. This field is available for image, and pushbutton dialog control types.

**Default:** empty

Container

Option to specifies a container for the child dialog control. The permitted values are the names of 'panel', 'group', and 'tab' dialog controls.
Examples

Add Dialog Control Elements to Mask Dialog Box

Get mask object and add dialog control element to it.

% Get mask object on model Engine

maskObj = Simulink.Mask.get('Engine/Gain');

% Add hyperlink to mask dialog box

maskLink = maskObj.addDialogControl('hyperlink','link');
maskLink.Prompt = 'Mathworks Home Page';
maskLink.Callback = 'web(''www.mathworks.com'')'

% Alternative method to add hyperlink

maskLink = maskObj.addDialogControl('hyperlink','link');
maskLink.Prompt = 'www.mathworks.com';

% Add text to mask dialog box

maskText = maskObj.addDialogControl('text','text_tag');
maskText.Prompt = 'Enable range checking';

% Add button to mask dialog box

maskButton = maskObj.addDialogControl('pushbutton','button_tag');
maskButton.Prompt = 'Compute';

Add Dialog Control Elements to Mask Dialog Box Tabs

Create tabs on the mask dialog box and add elements to these tabs.

% Get mask object on a block named 'GainBlock'

maskObj = Simulink.Mask.get('GainBlock/Gain');

% Create a tab container
maskObj.addDialogControl('tabcontainer','allTabs');
tabs = maskObj.getDialogControl('allTabs');

% Create tabs and name them
maskTab1 = tabs.addDialogControl('tab','First');
maskTab1.Prompt = 'First tab';

maskTab2 = tabs.addDialogControl('tab','Second');
maskTab2.Prompt = 'Second tab';

% Add elements to one of the tabs
firstTab = tabs.getDialogControl('First');
firstTab.addDialogControl('text','textOnFirst');
firstTab.getDialogControl('textOnFirst').Prompt = 'Tab one';

Add Dialog Control Element Using Name-Value Pair

Add dialog control element and specify values for it
% Get mask object on model Engine
maskObj = Simulink.Mask.get('Engine/Gain');
% Add a dialog box and specify values for it
maskDialog = maskObj.addDialogControl('Type','text',...
    'Prompt','hello','Visible','off');

See Also
Simulink.dialog.Container | “Create Block Masks”

Introduced in R2014a
removeDialogControl

Class: Simulink.dialog.Container
Package: Simulink.dialog

Remove dialog control element from mask dialog box

Syntax

successIndicator = maskVariable.removeDialogControl(controlIdentifier)

Description

successIndicator = maskVariable.removeDialogControl(controlIdentifier) removes dialog control element, specified by controlIdentifier, like text, hyperlinks, or tabs from a mask dialog box. First get the mask object and assign it to the variable maskVariable.

Successful removal of a dialog control element returns a Boolean value of 1.

Input Arguments

controlIdentifier — Unique identifier for the element
character vector

Programmatic identifier for the dialog control element of mask dialog box, specified as a character vector.

Examples

Remove Dialog Control Element from Mask Dialog Box

% Get mask object on the Gain block in the model Engine.
maskObj = Simulink.Mask.get('Engine/Gain');

% Remove element named AllTab from mask dialog box.
maskTab = maskObj.removeDialogControl('AllTab');

See Also
Simulink.dialog.Container | “Create Block Masks”

Introduced in R2013b
getDialogControl

**Class:** Simulink.dialog.Container  
**Package:** Simulink.dialog

Search for a specific dialog control on the mask dialog box.

**Syntax**

\[
[\text{control}, \text{phandle}] = \text{handle.getDialogControl(\text{controlIdentifier})}
\]

**Description**

\[
[\text{control}, \text{phandle}] = \text{handle.getDialogControl(\text{controlIdentifier})},
\]

search for a specific child dialog control recursively on the mask dialog box.

**Input Arguments**

\text{controlIdentifier}

Name of the dialog control being searched on the mask dialog box.

**Default:**

**Output Arguments**

\text{control}

Target dialog control being searched on the mask dialog box.

\text{phandle}

Parent of the dialog control being searched mask dialog box.
Examples

Find a dialog control

Find a text dialog control on the mask dialog box. `maskObj` is the handle to the mask object. The `getDialogControl` method returns the handle to the dialog control (`hdlgctrl`) and handle to the parent dialog control (`phandle`).

```matlab
[hdlgctrl, phandle] = maskObj.getDialogControl('txt_var')
```

See Also
Simulink.dialog.Container | “Create Block Masks”
Simulink.dialog.Panel class

**Package:** Simulink.dialog

Create an instance of a panel dialog control

**Description**

Use an instance of Simulink.dialog.Panel class to create an instance of panel dialog control.

**Properties**

**Name**

Uniquely identifies the panel dialog control and is a required field.

*Type:* character vector

**Row**

Specifies whether panel is placed on the current row or on a new row.

*Type:* character vector

*Values:* 'current'|'new'

*Default:* 'new'

**Enabled**

Specifies whether panel is active on the mask dialog box.

*Type:* boolean

*Values:* 'on'|'off'

*Default:* 'on'
**Visible**

Specifies whether panel is displayed on the mask dialog box.

**Type:** boolean

**Values:** 'on' | 'off'

**Default:** 'on'

**DialogControls**

Specifies the child dialog controls contained in the panel.

**Type:** Simulink.dialog.Control

**Default:** Empty array

**Methods**

- **addDialogControl**
  Add dialog control elements to mask dialog box

- **removeDialogControl**
  Remove dialog control element from mask dialog box

- **getDialogControl**
  Search for a specific dialog control on the mask

**See Also**

Simulink.dialog.Group | Simulink.dialog.Tab |
Simulink.dialog.TabContainer | Simulink.dialog.Container |
Simulink.dialog.Control | “Create Block Masks”
Simulink.dialog.Group class

Package: Simulink.dialog

Create an instance of a group dialog control

Description

Use an instance of Simulink.dialog.Group class to create an instance of group dialog control.

Properties

Name

Uniquely identifies the group dialog control and is a required field.

Type: character vector

Prompt

Specifies the text displayed on the group.

Type: character vector

Default: Empty character vector

Row

Specifies whether group is placed on the current row or on a new row.

Type: character vector

Values: 'current'|'new'

Default: 'new'
Enabled

Specifies whether group is active on the mask dialog box.

**Type:** boolean

**Values:** 'on' | 'off'

**Default:** 'on'

Visible

Specifies whether group is displayed on the mask dialog box.

**Type:** boolean

**Values:** 'on' | 'off'

**Default:** 'on'

DialogControls

Specifies the child dialog controls contained in the group.

**Type:** Simulink.dialog.Control

**Default:** Empty array

Methods

- addDialogControl: Add dialog control elements to mask dialog box
- removeDialogControl: Remove dialog control element from mask dialog box
- getDialogControl: Search for a specific dialog control on the mask

See Also

- Simulink.dialog.Panel
- Simulink.dialog.Tab
- Simulink.dialog.TabContainer
- Simulink.dialog.Container
- Simulink.dialog.Control
- “Create Block Masks”
Simulink.dialog.Tab class

**Package:** Simulink.dialog

Create an instance of a tab dialog control

**Description**

Use an instance of Simulink.dialog.Tab class to create an instance of tab dialog control.

**Properties**

**Name**

Uniquely identifies the tab dialog control and is a required field.

**Type:** character vector

**Prompt**

Specifies the text displayed on the tab.

**Type:** character vector

**Default:** Empty character vector

**Enabled**

Specifies whether tab is active on the mask dialog box.

**Type:** boolean

**Values:** 'on'|'off'

**Default:** 'on'
Visible

Specifies whether tab is displayed on the mask dialog box.

**Type:** boolean

**Values:** 'on'|'off'

**Default:** 'on'

DialogControls

Specifies the child dialog controls contained in the tab dialog control.

**Type:** Simulink.dialog.Control

**Default:** Empty array

**Methods**

- addDialogControl
  - Add dialog control elements to mask dialog box
- removeDialogControl
  - Remove dialog control element from mask dialog box
- getDialogControl
  - Search for a specific dialog control on the mask

**See Also**

Simulink.dialog.TabContainer class

**Package:** Simulink.dialog

Create an instance of a tab container dialog control

**Description**

Use an instance of `Simulink.dialog.TabContainer` class to create an instance of tab container dialog control. Tab container dialog box be used to group the tab dialog controls.

**Properties**

**Name**

Uniquely identifies the tab container dialog control and is a required field.

**Type:** character vector

**Row**

Specifies whether tab container is placed on the current row or on a new row.

**Type:** enumerated string

**Values:** 'current'|'new'

**Default:** 'new'

**Enabled**

Specifies whether tab container is active on the mask dialog box.

**Type:** boolean

**Values:** 'on'|'off'

**Default:** 'on'
Visible

Specifies whether tab container is displayed on the mask dialog box.

**Type:** boolean

**Values:** 'on' | 'off'

**Default:** 'on'

DialogControls

Specifies the child dialog controls contained in the group. Simulink.dialog.TabContainer class can only contain Simulink.dialog.Tab dialog control.

**Type:** Simulink.dialog.Tab

**Default:** Empty array

Methods

- addDialogControl: Add dialog control elements to mask dialog box
- removeDialogControl: Remove dialog control element from mask dialog box
- getDialogControl: Search for a specific dialog control on the mask

See Also

Simulink.dialog.Button class

**Package:** Simulink.dialog

Create a button dialog control

**Description**

Use an instance of `Simulink.dialog.Button` class to add a button dialog control.

**Properties**

**Name**

Uniquely identifies the dialog control and is a required field.

**Type:** character vector

**Prompt**

Specifies the text displayed on the button dialog control.

**Type:** character vector

**Default:** empty

**FilePath**

Specifies the path to the image file to be shown on the button dialog control.

**Type:** character vector

**Default:** empty

**Callback**

Specifies the MATLAB command(s) to be executed when the dialog control is invoked.

**Type:** character vector
**Default:** empty

**Row**

Specifies whether dialog control is placed on the current row or on a new row.

**Type:** character vector

**Value:** 'current' | 'new'

**Default:** 'current'

**Enabled**

Indicates whether container is active on the mask dialog box.

**Type:** boolean

**Value:** 'on' | 'off'

**Default:** 'on'

**Visible**

Indicates whether container is displayed on the mask dialog box.

**Type:** boolean

**Value:** 'on' | 'off'

**Default:** 'on'

**See Also**

Simulink.dialog.Control | “Create Block Masks”
Simulink.dialog.Hyperlink class

**Package:** Simulink.dialog

Create a hyperlink dialog control

**Description**

Use an instance of `Simulink.dialog.Hyperlink` class to add a hyperlink dialog control.

**Properties**

**Name**

Uniquely identifies the dialog control and is a required field.

*Type:* character vector

**Prompt**

Specifies the text displayed on the hyperlink.

*Type:* character vector

*Default:* empty

**Callback**

Specifies the MATLAB command (s) to be executed when the dialog control is invoked.

*Type:* character vector

*Default:* empty

**Row**

Specifies whether hyperlink is placed on the current row or on a new row.
Type: character vector
Value: 'current'|'new'
Default: 'new'

Enabled
Indicates whether hyperlink is active on the mask dialog box.
Type: boolean
Value: 'on'|'off'
Default: 'on'

Visible
Indicates whether hyperlink is displayed on the mask dialog box.
Type: boolean
Value: 'on'|'off'
Default: 'on'

See Also
Simulink.dialog.Control | “Create Block Masks”
Simulink.dialog.Image class

**Package:** Simulink.dialog

Create an image dialog control

**Description**

Use an instance of Simulink.dialog.Image class to add an image dialog control.

**Properties**

- **Name**
  Uniquely identifies the dialog control and is a required field.
  
  **Type:** character vector

- **FilePath**
  Specifies the path to the image file to be displayed on the dialog box.
  
  **Type:** character vector
  
  **Default:** empty

- **Row**
  Specifies whether dialog control is placed on the current row or on a new row.
  
  **Type:** character vector
  
  **Value:** 'current' | 'new'
  
  **Default:** 'new'

- **Enabled**
  Indicates whether image is active on the mask dialog box.
Type: boolean

Value: 'on'|'off'

Default: 'on'

Visible

Indicates whether image is displayed on the mask dialog box.

Type: boolean

Value: 'on'|'off'

Default: 'on'

See Also

Simulink.dialog.Control | “Create Block Masks”
Simulink.dialog.Text class

Package: Simulink.dialog

Create a text dialog control

Description

Use an instance of Simulink.dialog.Text class to add a text dialog control.

Properties

Name
Uniquely identifies the dialog control element and is a required field.
Type: character vector

Prompt
Specifies the text displayed on the mask dialog box.
Type: character vector
Default: empty

WordWrap
Specifies whether to wrap long text to the next line on the mask dialog box.
Type: boolean
Value: 'on'|'off'
Default: 'on'

Row
Specifies whether dialog control is placed on the current row or on a new row.
**Type:** character vector  
**Value:** 'current' | 'new'  
**Default:** 'new'

**Enabled**  
Indicates whether dialog control is active on the mask dialog box.  
**Type:** boolean  
**Value:** 'on' | 'off'  
**Default:** 'on'

**Visible**  
Indicates whether dialog control is displayed on the mask dialog box.  
**Type:** boolean  
**Value:** 'on' | 'off'  
**Default:** 'on'

**See Also**  
Simulink.dialog.Control | “Create Block Masks”
Simulink.dialog.parameter.Control class

**Package:** Simulink.dialog.parameter

Create a parameter dialog control

**Description**

Use an instance of Simulink.dialog.parameter.Control class to add a parameter dialog control.

**Properties**

**Name**

Uniquely identifies the dialog control element. This is a required field and has the same value as its underlying parameter name.

**Type:** character vector

**Row**

Specifies whether the dialog control is placed on the current row or on a new row.

**Type:** character vector

**Value:** 'current'|'new'

**Default:** 'new'

**See Also**

Simulink.dialog.Control | “Create Block Masks”
Simulink.op.ModelOperatingPoint class

Package: Simulink.op

Access simulation operating point data

Description

The Simulink.op.ModelOperatingPoint class contains all of the information associated with an operating point of a simulation, including the logged states, the time of the snapshot, and the start time of the simulation. To access this data for a specific block in the model, use the `get` method or the `loggedStates` property.

Properties

description — Description of the saved operating point
class: character vector

Description of the saved operating point object, specified as a character vector. By default, Simulink generates a character vector based on your model name.

loggedStates — Continuous and discrete states of the blocks
class: dataset | structure

Continuous and discrete states of the block that depend on the `SaveFormat` settings of the model.

If specified as a `Dataset` format, you cannot assign a structure or a `Simulink.SimulationData.Dataset` object with a different number of elements than that of the `Dataset` object used for `loggedStates`.

If specified as a `Structure` format, you cannot assign a `Dataset` object.

snapshotTime — Simulation time of saving operating point
class: double

Time at which Simulink takes a snapshot of the complete simulation operating point.
This data is read-only.

**startTime — Time at which simulation starts**

double

Time at which simulation of the model starts.

This data is read-only.

## Methods

### Public Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>set</td>
<td>Set operating point of individual Stateflow chart, MATLAB function, or S-function block</td>
</tr>
<tr>
<td>get</td>
<td>Access operating point of individual Stateflow chart, MATLAB function, or S-Function block</td>
</tr>
</tbody>
</table>

## Examples

### Save and Restore an Operating Point

Use the `sim` command with `set_param`. Set the `SaveOperatingPoint` parameter to 'on'.

```matlab
fuelsys
set_param('fuelsys','SaveFinalState','on','FinalStateName',...
'myOperPoint','SaveOperatingPoint','on');
simOut = sim('fuelsys','StopTime','10')
myOperPoint = simOut.myOperPoint
```

**Note** Before you save the operating point, it is recommended that you disable the **Block Reduction** parameter in **Configuration Settings > Simulation Target > Advanced Parameters**.
See Also
get | set

Topics
“Save and Restore Simulation Operating Point”
“Operating Point Behavior”

Introduced in R2019a
**get**

**Class:** Simulink.op.ModelOperatingPoint  
**Package:** Simulink.op

Access operating point of an individual Stateflow chart, MATLAB function, or S-function block

**Syntax**

blockOperatingPoint = get(modelOperatingPoint, 'blockpath')

**Description**

blockOperatingPoint = get(modelOperatingPoint, 'blockpath') returns the operating point of the block specified by blockpath. The blockpath must refer to a Stateflow chart, MATLAB functions, or S-function block. To access states of other types of blocks, see the loggedStates property of the ModelOperatingPoint class.

**Input Arguments**

- **modelOperatingPoint** — A Simulink.op.ModelOperatingPoint object
  
  Operating point object for the model.

- **blockpath** — Path of the block
  
  Character vector

  Path of the block to set the operating point values, specified as a character vector

**Output Arguments**

- **blockOperatingPoint** — Operating point of the block specified
  
  String
Name of the operating point to be set to the specified block.

**Examples**

**Get the Operating Point of a Block**

To get the operating point of a block that is in a referenced model, specify the full path of the block relative to the root model.

```matlab
model = 'sldemo_fuelsys_dd';
opt = struct('SaveFinalState','on','SaveOperatingPoint','on','StopTime','1');
simOut = sim(model,opt);
modelOp = simOut.modelOpFinal;
blockPath = 'sldemo_fuelsys_dd/Fuel Rate Controller|sldemo_fuelsys_dd_controller/control_logic';
chartOp = get(modelOp, blockPath)
```

This data is read only.

**See Also**

Simulink.op.ModelOperatingPoint | set | ssSetOperatingPointVisibility

**Topics**

“Save and Restore Simulation Operating Point”
“Operating Point Behavior”

**Introduced in R2019a**
set

**Class:** Simulink.op.ModelOperatingPoint  
**Package:** Simulink.op

Set operating point of an individual Stateflow chart, MATLAB function, or S-function block.

**Syntax**

```
set(modelOperatingPoint,'blockpath',blockOperatingPoint)
```

**Description**

`set(modelOperatingPoint,'blockpath',blockOperatingPoint)` sets the operating point to the block specified by a `blockpath`. The `blockpath` must refer to a Stateflow chart, a MATLAB function, or a S-function block. To modify states of other types of blocks, use the `loggedStates` of the `Simulink.op.ModelOperatingPoint` class.

**Input Arguments**

- **modelOperatingPoint** — A `Simulink.op.ModelOperatingPoint` object  
  
  Operating point object for the model.

- **blockpath** — Path of the block  
  
  Character vector  
  
  Path of the block for which to set the operating point values, specified as a character vector.

- **blockOperatingPoint** — Operating point of the block specified  
  
  Object  
  
  Operating point to be set to the specified block, specified as an object.
Examples

Set the Operating Point of a Block

This example shows how to set the operating point for a block.

\[
\text{newOperPoint} = \text{set}(\text{myOperPoint}, \text{'mymodel/chart'}, \text{newChartOperPoint})
\]

See Also

Simulink.op.ModelOperatingPoint | get | ssSetOperatingPointVisibility

Topics

“Save and Restore Simulation Operating Point”
“Operating Point Behavior”

Introduced in R2019a
Simulink.sfunction.Analyzer class

Package: Simulink.sfunction

Create a Simulink S-function analyzer object

Description

This class enables you to perform checks on S-functions within a model or a library. These checks include MEX compiler setup check, source code check, MEX-file check, parameter robustness check for S-functions. The check result can be accessed either from a MATLAB structure or an HTML report.

The S-function analyzer checks the source code of the S-functions based on the S-function names. The S-function source code can be automatically included in the analysis if the source file is a single .c or .cpp file in the MATLAB path that has the same name as the S-function. Otherwise, the build information can be specified through the S-function Analyzer APIs. If no source code is available on the specified path, the analysis is skipped.

Construction

sfunAnalyzer = Simulink.sfunction.Analyzer(model) creates a Simulink.sfunction.Analyzer object with the model you specify. In this case, the source code for the S-function can be automatically included in the analysis if the source code file is a single .c or .cpp file in the MATLAB path that has the same name as the S-function. For example, if the specified model contains an S-function called mysfun, and the source file for mysfun is a single file mysfun.c in the MATLAB path, a Simulink.sfunction.analyzer.BuildInfo object is automatically created and included in the analysis.

sfunAnalyzer = Simulink.sfunction.Analyzer(model,'BuildInfo',{bdInfo}) creates a Simulink.sfunction.Analyzer object with the model and a Simulink.sfunction.analyzer.BuildInfo object named bdInfo.

sfunAnalyzer = Simulink.sfunction.Analyzer(model,'Options',{opts}) creates an Simulink.sfunction.Analyzer object with the model and a Simulink.sfunction.analyzer.Options object named opts.
Input Arguments

**model** — Specify a model in the path
character vector | string

Names of the model in the path, specified as a string or character vector.

Name-Value Pair Arguments

Specify optional comma-separated pairs of **Name**, **Value** arguments. **Name** is the argument name and **Value** is the corresponding value. **Name** must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'BuildInfo', {buildInfo}'

**BuildInfo** — Specify the buildinfo object
cell array

A cell array of buildinfo objects. See `Simulink.sfunction.analyzer.BuildInfo` for more information.

**Options** — Specify the S-function analyzer running options
object

An object to specify S-function analyzer running options. These checks include Polyspace® Code Prover™ and parameter robustness checks. See `Simulink.sfunction.analyzer.Options` for more information.

Methods

- **run** Perform checks on S-functions
- **generateReport** Generate an HTML report of S-function checks

See Also

- `Simulink.sfunction.Analyzer.generateReport`
- `Simulink.sfunction.Analyzer.run`
- `Simulink.sfunction.analyzer.BuildInfo`
Simulink.sfunction.analyzer.Options
Simulink.sfunction.analyzer.findSfunctions

Introduced in R2017b
run

Class: Simulink.sfunction.Analyzer
Package: Simulink.sfunction

Perform checks on S-functions

Syntax

result = run()

Description

result = run() returns a struct containing the result from the analyzer checks. An example result struct has the following fields:

- Platform: 'win64'
- Release: '(R2017b)'
- SimulinkVersion: '9.0'
- ExemptedBlocks: {}
- MexConfiguration: [1×1 mex.CompilerConfiguration]
  Data: [4×4 struct]

Output Arguments

result — Sfunction.Analyzer structure
MATLAB structure

MATLAB structure containing the result of S-function analyzer.

See Also

Simulink.sfunction.Analyzer|
Simulink.sfunction.Analyzer.generateReport|
Simulink.sfunction.analyzer.BuildInfo |
Simulink.sfunction.analyzer.Options |
Simulink.sfunction.analyzer.findSfunctions

Introduced in R2017b
generateReport

**Class:** Simulink.sfunction.Analyzer  
**Package:** Simulink.sfunction

Generate an HTML report of S-function checks

**Syntax**

```matlab
generateReport()
```

**Description**

generateReport() generates an HTML report and launches the browser to display the report.

**See Also**


**Introduced in R2017b**
Simulink.sfunction.analyzer.BuildInfo class

Package: Simulink.sfunction.analyzer

Create an object to represent build information

Description

Simulink.sfunction.analyzer.BuildInfo object captures the build information for S-functions, such as source files, header files, and linking libraries, for use with the Simulink.sfunction.Analyzer class.

Construction

bdInfo = Simulink.sfunction.analyzer.BuildInfo(SfcnFile) creates a Simulink.sfunction.analyzer.BuildInfo object.


bdInfo = Simulink.sfunction.analyzer.BuildInfo(SfcnFile, 'ObjFileList',{objfilelist}) creates a Simulink.sfunction.analyzer.BuildInfo object for C-MEX S-function source file and list of extra objective code files.

bdInfo = Simulink.sfunction.analyzer.BuildInfo(SfcnFile, 'IncPaths', {incpathsl}), 'LibFileList'
creates a Simulink.sfunction.analyzer.BuildInfo object for C-MEX S-function source file and paths to the folders including header files.


bdInfo = Simulink.sfunction.analyzer.BuildInfo(SfcnFile, 'PreProcDefList',{preprocdir}) creates a
Simulink.sfunction.analyzer.BuildInfo object for C-MEX S-function source file and pre-processor directives list.

**Input Arguments**

**SfcnFile — S-function source file**
character vector | string

S-function source file having the same name as the S-function.

**Name-Value Pair Arguments**

Specify optional comma-separated pairs of Name, Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside quotes. You can specify several name and value pair arguments in any order as Name1, Value1, ..., NameN, ValueN.

Example: 'SrcPath', {srcpath}

**SrcPath — Source file path**
cell array of character vectors | string

Search paths to extra S-function source files that are referenced by SfcnFile, specified as a cell array or a string.

**ExtraSrcFileList — Extra source file list**
cell array of character vectors | string

List of extra S-function source files, specified as a cell array or string.

**ObjFileList — Extra objective code**
cell array

List of objective files used for building, specified as a cell array.

**IncPaths — Search paths for header files**
cell array of character vectors | string

Specify paths to include folders for header files, specified as a cell array or a string.

**LibFileList — External libraries list**
cell array of character vectors | string
List of external library files used for building, specified as a cell array or string.

**LibPaths — Search paths for external libraries**
cell array of character vectors | string

Search paths for external library files used for building, specified as a cell array or a string.

**PreProcDefList — Preprocessor directives**
cell array of character vectors | string

List of the preprocessor directives, specified as a cell array or a string.

**Output Arguments**

**bdInfo — BuildInfo object**
object

Build information for the S-functions supplied to the S-function analyzer. Returns a *simulink.sfunction.analyzer.BuildInfo* object.

**Examples**

Create a bdInfo object for an S-function mysfun that includes a source file mysfun.c:

**Basic Use**

bdInfo = Simulink.sfunction.analyzer.BuildInfo('mysfun.c');

The output bdInfo has the following fields:

bdInfo =

BuildInfo with properties:

SfcnFile: 'mysfun.c'
SfcnName: 'mysfun'
SrcType: 'C'
SrcPaths: {]
Create a `bdInfo` object for an S-function `mysfun` that includes a source file `mysfun.c` and also includes:

- List of extra source files, `extra1.c` and `extra2.c`
- Paths to source file folders, `/path1` and `/path2`.
- List of objective files, `o1.obj` and `o2.obj`.
- List of library files, `l1.lib` and `l2.lib`.
- Library paths, `/libpath1`.
- Pre-processor running directives, `-DDEBUG`.

```matlab
Simulink.sfunction.analyzer.BuildInfo('mysfun.c', ...
    'ExtraSrcFileList',{extra1.c,extra2.c},...
    'SrcPaths',{/path1,/path2},...
    'ObjFileList',{o1.obj,o2.obj},...
    'LibFileList',{l1.lib,l2.lib},...
    'LibPaths',{/libpath1},...
    'PreProcDefList',{DEBUG});
```

### See Also

- `Simulink.sfunction.Analyzer`
- `Simulink.sfunction.Analyzer.generateReport`
- `Simulink.sfunction.Analyzer.run`
- `Simulink.sfunction.analyzer.Options`
- `Simulink.sfunction.analyzer.findSfunctions`

**Introduced in R2017b**
Simulink.sfunction.analyzer.Options class

Package: Simulink.sfunction.analyzer

Create an object to specify options for running S-function checks

Description

Simulink.sfunction.analyzer.Options object is created through the constructor
Simulink.sfunction.analyzer.Options().
Simulink.sfunction.analyzer.Options object captures the options for running S-
function checks. These checks include whether to enable Polyspace and Parameter
Robustness checks, maximum model simulation time and output path for result report.

Construction

opts= Simulink.sfunction.analyzer.Options() returns a options object with
these property values:

EnablePolyspace: 0
EnableRobustness: 0
ReportPath: ''
ModelSimTimeOut: 10

Properties

EnablePolyspace — Polyspace Code Prover check
False (default) | True

Boolean type check to determine whether to include Polyspace Code Prover check.

Note These checks usually take some time to run.
EnableRobustness — Parameter robustness check
False (default) | True

Boolean type check to indicate whether to include Robustness checks.

**Note** These checks usually take some time to run.

ReportPath — Generated report directory
current working directory (default) | character array

Path to the generated report directory.

ModelSimTimeOut — Maximum model simulation time
10 (default) | scalar

Maximum model simulation time in seconds.

EnableUsePublishedOnly — Check for use of only documented APIs
False (default) | True

Check to indicate if any undocumented S-function APIs are used in the code.

**See Also**
Simulink.sfunction.Analyzer
Simulink.sfunction.Analyzer.generateReport
Simulink.sfunction.Analyzer.run
Simulink.sfunction.analyzer.BuildInfo
Simulink.sfunction.analyzer.findSfunctions

**Introduced in R2017b**
findSfunctions

Find and return all eligible S-functions in a model

Syntax

sfuns = Simulink.sfunction.analyzer.findSfunctions(model)

Description

sfuns = Simulink.sfunction.analyzer.findSfunctions(model) returns all eligible S-functions in a model for the S-function checks. Rules are applied to filter out all ineligible S-functions.

Input Arguments

model — A Simulink model or library in path
character vector | string vector

A Simulink model or library in path specified as a string or a character vector.

Output Arguments

sfuns — A list of all eligible S-functions
cell array of character vectors

Eligible S-functions in the model, specified as a cell array of character vectors.

See Also
Simulink.sfunction.Analyzer
Simulink.sfunction.Analyzer.generateReport
Simulink.sfunction.Analyzer.run
Simulink.sfunction.analyzer.BuildInfo
Simulink.sfunction.analyzer.Options

Introduced in R2017b
plot

Package: Simulink.SimulationData

Plot simulation output data in the Simulation Data Inspector

Syntax

plot(simOutObj)
runObj = plot(simOutObj)

Description

plot(simOutObj) plots the simulation output data in the simulation output object, simOutObj, in the Simulation Data Inspector and opens the Simulation Data Inspector so you can view the plotted simulation output data. You can use the plot function to plot simulation results stored in these simulation output objects:

• Simulink.SimulationOutput
• Simulink.SimulationData.DataStoreMemory
• Simulink.SimulationData.Parameter
• Simulink.SimulationData.Signal
• Simulink.SimulationData.State
• sltest.Assessment

These simulation output objects also have plot functions that plot the data in and then open the Simulation Data Inspector:

• Simulink.SimulationData.Dataset
• Simulink.SimulationData.DatasetRef

When a simulation output object contains fewer than eight signals to plot, the Simulation Data Inspector layout changes to 1-by-n, where n is the number of signals to plot, and plots one signal on each subplot. When the simulation output object contains more than
eight signals to plot, the Simulation Data Inspector layout changes to 1-by-1 and plots the first signal in the simulation output object.

When some or all of the data in a Simulink.SimulationOutput object is in a Simulation Data Inspector run, the plot function opens the Simulation Data Inspector and plots all the signals in the run. When you do not select the Record logged workspace data in Simulation Data Inspector option, logged states data does not appear in the Simulation Data Inspector and is not plotted. When the data does not correspond to a run in the Simulation Data Inspector, the plot function imports the data to a new run. When you use the plot function to plot a single signal, the plot function always imports the data for the signal to a new run.

runObj = plot(simOutObj) returns the Simulink.sdi.Run object corresponding to the plotted data.

**Examples**

**Access and Plot Simulation Output Data**

When you create a new model in R2019a or later, the default model configuration saves simulation outputs in a single output. The single simulation output is a Simulink.SimulationOutput object that contains one or more types of other simulation output objects, depending on the kinds of data you log. This example uses a model configured to save a single simulation output and shows how to access each type of logged data and use the plot function to plot the data in the Simulation Data Inspector and then open the Simulation Data Inspector so you can view the data.

The ex_vdp_simout_plot model used in this example is configured to log signals, outputs, and states and return all logged data in a single simulation output. This example shows how to access each type of logged data and use the plot function to plot the data in the Simulation Data Inspector. To generate the SimulationOutput object containing all the logged data, simulate the model.

open_system('ex_vdp_simout_plot.slx')
out = sim('ex_vdp_simout_plot.slx');

**Plot Data in a SimulationOutput Object**

You can pass the single simulation output, stored in a Simulink.SimulationOutput object, to the plot function to plot and view the data in the Simulation Data Inspector.
When you plot data in a SimulationOutput object that corresponds to a run in the Simulation Data Inspector, data in the object that also logs to the Simulation Data Inspector is plotted. The model logs four signals, data for the two Outport blocks, and states. Signal and output data always log to the Simulation Data Inspector, and states data only appears in the Simulation Data Inspector when you select the **Record logged workspace data in Simulation Data Inspector** option, which is not selected for this model. When you use the `plot` function to plot the data, the Simulation Data Inspector layout updates to a 1-by-6 layout and plots one signal on each subplot.

```matlab
plot(out)
```

**Plot Logged States Data**

When you do not log states data to the Simulation Data Inspector, you can use the `plot` function to import and plot the states data. The model logs data for two states, `x1` and `x2`. The states data is saved in a `Simulink.SimulationData.Dataset` object, `xout`, with one element corresponding to each `Simulink.SimulationData.State` object. You can use the `plot` function to plot the data for both signals by plotting `xout`, or you can plot data for a single state.

Access the Dataset object, `xout`, using the `get` function for the `SimulationOutput` object. You can also access the logged output and signal data using the `get` function.

```matlab
xout = get(out,'xout');
```

When you plot the data for both states in the Dataset object, the Simulation Data Inspector layout changes to 1-by-2 and plots the data for each state on one subplot.

```matlab
plot(xout)
```

**Plot Data for a Single Signal**

When you plot the data for a single signal, the Simulation Data Inspector always imports the data for the signal to a new run. Use the `get` function for the `SimulationOutput` object to access the signal logging Dataset object, `logsout`.

```matlab
logsout = get(out,'logsout');
```

Then, use the `get` function for the Dataset object to access the data for the first element.

```matlab
sig1 = get(logsout,1);
```
When you plot the data for the signal, the Simulation Data Inspector imports the signal to a new run, updates the layout to 1-by-1, and plots the signal.

`plot(sig1)`

**Input Arguments**

`simOutObj` — Simulation output object containing simulation data to plot


Simulation output object containing data you want to plot and view in the Simulation Data Inspector. This `plot` function supports these simulation output objects:

- `Simulink.SimulationOutput`
- `Simulink.SimulationData.DataStoreMemory`
- `Simulink.SimulationData.Parameter`
- `Simulink.SimulationData.Signal`
- `Simulink.SimulationData.State`
- `sltest.Assessment`

Example: `plot(out)`

**Output Arguments**

`runObj` — Simulink.sdi.Run object corresponding to plotted data

`Simulink.sdi.Run` object

`Simulink.sdi.Run` object corresponding to the plotted data.

**See Also**

`plot`

**Topics**

“View Data with the Simulation Data Inspector”
Introduced in R2019b
addElement

**Class:** Simulink.SimulationData.Dataset  
**Package:** Simulink.SimulationData

Add element to end of data set

**Syntax**

dataset = addElement(dataset,element)  
dataset = addElement(dataset,element,name)

**Description**

dataset = addElement(dataset,element) adds an element to the Simulink.SimulationData.Dataset dataset.

dataset = addElement(dataset,element,name) adds an element to the Simulink.SimulationData.Dataset data set and gives the element the name that you specify with the name argument. If the object already has a name, the element instead uses the name you specify by using the name argument.

**Input Arguments**

*dataset — Data set*  
SimulationData.Dataset object

The data set to which to add the element.

*element — Element to add*  

**name** — Name for element
character vector

Name for element, specified as a character vector.

## Output Arguments

**dataset** — Data set
character vector

The data set to which you add the element, returned as a character vector. The new element is added to the end of the data set.

## Examples

### Create a Data Set

Create a data set and add three elements to it.

```matlab
time = 0.1*(0:100)';
ds = Simulink.SimulationData.Dataset;
element1 = Simulink.SimulationData.Signal;
element1.Name = 'A';
element1.Values = timeseries(sin(time),time);
ds = addElement(ds,element1);
element2 = Simulink.SimulationData.Signal;
element2.Name = 'B';
element2.Values = timeseries(2*sin(time),time);
ds = addElement(ds,element2);
element3 = Simulink.SimulationData.Signal;
element3.Name = 'C';
element3.Values = timeseries(3*sin(time),time);
ds = addElement(ds,element3);
ds```

5-1234
ds = Simulink.SimulationData.Dataset '' with 3 elements

<table>
<thead>
<tr>
<th>Name</th>
<th>BlockPath</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 [1x1 Signal]</td>
<td>A ''</td>
</tr>
<tr>
<td>2 [1x1 Signal]</td>
<td>B ''</td>
</tr>
<tr>
<td>3 [1x1 Signal]</td>
<td>C ''</td>
</tr>
</tbody>
</table>

- Use braces {} to access, modify, or add elements using index.

**Alternative**

To streamline indexing syntax, you can use curly braces ({{}}) to add an element to a dataset, instead of using `addElement`. For the index, use a scalar that is greater than the number of elements by one. The new element becomes the last element of the dataset.

```plaintext
time = 0.1*(0:100)';
ds = Simulink.SimulationData.Dataset;
element1 = Simulink.SimulationData.Signal;
element1.Name = 'A';
element1.Values = timeseries(sin(time),time);
ds{1} = element1;
element2 = Simulink.SimulationData.Signal;
element2.Name = 'B';
element2.Values = timeseries(2*sin(time),time);
ds{2} = element2;
element3 = Simulink.SimulationData.Signal;
element3.Name = 'C';
element3.Values = timeseries(3*sin(time),time);
ds{3} = element3;
```

**See Also**

Simulink.SimulationData.BlockPath |
Simulink.SimulationData.DataStoreMemory |
Simulink.SimulationData.Dataset |
Simulink.SimulationData.Dataset.concat |
Simulink.SimulationData.Dataset.find |
Simulink.SimulationData.Dataset.get |
Simulink.SimulationData.Dataset.getElementNames |
Simulink.SimulationData.Dataset.numElements |
Simulink.SimulationData.Dataset.setElement |
Simulink.SimulationData.Signal |
matlab.io.datastore.SimulationDatastore  

**Topics**

“Migrate Scripts That Use Legacy ModelDataLogs API”
“Export Signal Data Using Signal Logging”
“Log Data Stores”
“Convert Logged Data to Dataset Format”
“Migrate Scripts That Use Legacy ModelDataLogs API”

**Introduced in R2011a**
concat

Class: Simulink.SimulationData.Dataset
Package: Simulink.SimulationData

Concatenate dataset to another dataset

Syntax

dataset1 = concat(dataset1,dataset2)

Description

dataset1 = concat(dataset1,dataset2) concatenates the elements of dataset2 to dataset1.

Input Arguments

dataset1 — Dataset to concatenate to
data set

Dataset to concatenate to with dataset2, returned as a cell array.

dataset2 — Dataset to concatenate
data set

Data set to concatenate to dataset1, specified as a cell array.

Output Arguments

dataset1 — Concatenated dataset
data set

Concatenated dataset from dataset1 and dataset2.
Examples

Concatenate ds1 to ds

Convert output from two To Workspace blocks to Dataset format and concatenate them.

mdl = 'myvdp';
open_system(mdl);
sim(mdl)
ds = Simulink.SimulationData.Dataset(simout);
ds1 = Simulink.SimulationData.Dataset(simout1);
rsfinal = concat(ds,ds1);

See Also


Topics

“Migrate Scripts That Use Legacy ModelDataLogs API”
“Export Signal Data Using Signal Logging”
“Log Data Stores”
“Convert Logged Data to Dataset Format”
“Migrate Scripts That Use Legacy ModelDataLogs API”

Introduced in R2015a
exportToPreviousRelease

Package: Simulink.SimulationData

Save a Dataset object to a MAT-file you can open in any release

Syntax

exportToPreviousRelease(ds,filename,dsname)
exportToPreviousRelease(____,version)
exportToPreviousRelease(____,'–append')
exportToPreviousRelease(____,'–nocompression')

Description

exportToPreviousRelease(ds,filename,dsname) saves the Simulink.SimulationData.Dataset object, ds, to the MAT-file specified by the filename input, using the name specified by the dsname input. You can load the Dataset saved in the file in any release that supports the Dataset format. When you do not use this function to save a Dataset object to a file, you may not be able to load the data in releases prior to R2017a. When you do not need to open and use the Dataset data in a release prior to R2017a, you can use the save function.

exportToPreviousRelease(____,version) saves data stored in a Dataset object to a MAT-file you can load in a previous release, using the MAT-file version specified by version.

exportToPreviousRelease(____,'–append') adds data stored in a Dataset object to a MAT-file you can load in a previous release without overwriting the contents of the MAT-file.

exportToPreviousRelease(____,'–nocompression') saves the Dataset object to a MAT-file without compression. The '–nocompression' option only supports Version 7 and Version 7.3 MAT-files.
Examples

Save a Dataset to a MAT-File to Load in a Prior Release

You can use the `exportToPreviousRelease` function to save data stored in a `Simulink.SimulationData.Dataset` object in a MAT-file that you can load in a previous release. This example saves data logged from a simulation in a MAT-file using the `exportToPreviousRelease` function, and then appends another `Dataset` object containing data logged from another simulation to the same MAT-file.

**Save Dataset Data to a MAT-File**

To create a `Dataset` object containing logged data, run a simulation of the `ex_vdp` model that logs data for signals `x1` and `x2`. The model is configured to generate a single simulation output containing all simulation data, including the signal logging data stored in the `Dataset` object, `logsout`.

```matlab
open_system('ex_vdp.slx')
out = sim('ex_vdp');
logsout = out.logsout;
```

Use the `exportToPreviousRelease` function to save the signal logging data to a MAT-file you can share with colleagues and collaborators who may use different Simulink™ versions.

```matlab
exportToPreviousRelease(logsout,'vdpLoggedData.mat','ex_vdpSim1')
```

**Append Additional Data to the MAT-File**

You can use the `exportToPreviousRelease` function to add data to a MAT-file. For example, if you run another simulation that logs data and get new results you want to share. You can load the `Dataset` object saved using the `exportToPreviousRelease` function in any release that supports the `Dataset` format.

```matlab
set_param('ex_vdp/Mu','Gain','0.5')
out = sim('ex_vdp');
logsout = out.logsout;
```
exportToPreviousRelease(logsout,'vdpLoggedData.mat','ex_vdpSim2','-append')

**Input Arguments**

**ds** — Dataset object containing data you want to load in an earlier release

Simulink.SimulationData.Dataset object

Simulink.SimulationData.Dataset object you want to save in a MAT-file and load in a previous release.

Example: logsout

**filename** — Name of file to export data to

string | character array

Name of the file to export data to. If the file does not exist, the `exportToPreviousRelease` function creates the file. When the file exists, you can overwrite the data in the file, or you can use the `-append` input to append the Dataset data to the file.

Example: 'myfile.mat'

Data Types: char | string

**dsname** — Name to use for the saved Dataset object in the MAT-file

string | character array

Name to use for the Dataset object saved in the MAT-file. The name does not have to match the name of the Dataset object containing the data you want to save in the MAT-file.

Example: 'myLoggedData'

Data Types: char | string

**version** — MAT-file version

'-v7' (default) | '-v7.3' | '-v6'

MAT-file version. By default, the `exportToPreviousRelease` function creates Version 7 MAT-files, unless you have changed the preference for the default MAT-file version created by save operations. For more information, see “MAT-File Versions” (MATLAB). You cannot save Dataset data to Version 4 MAT-files.
Data Types: `char` | `string`

**See Also**

`Simulink.SimulationData.Dataset` | `save`

**Topics**

“MAT-File Versions” (MATLAB)

**Introduced in R2019b**
get

Class: Simulink.SimulationData.Dataset
Package: Simulink.SimulationData

Get element or collection of elements from dataset

Syntax

```
element = get(dataset,index)
element = get(dataset,name)
element = get(dataset,{name})
```

Description

- `element = get(dataset,index)` returns the element corresponding to the index. The `getElement` method uses the same syntax and behavior as the get method.

- `element = get(dataset,name)` returns the element whose name matches `name`. When `name` is in a cell array, return the index of the element whose name matches `name`.

- `element = get(dataset,{name})` returns a single element if only one element name matches, a `SimulationData.Dataset` if multiple elements with this name exist.

If you use Log Dataset data to file to create the MAT-file, use `getAsDatastore` for fast access to the data.

Input Arguments

- **dataset** — Dataset
  SimulationData.Dataset object
  The data set from which to get the element.

- **index** — Index value of element to get
  scalar numeric
Index value of element to get. The index reflects the index value of a data set element.

\textbf{name — Name for data set element}

\texttt{character array | cell array}

Name for a data set element, specified as:

- A character array reflecting the name of the data set element
- A cell array containing one character vector. To return a \texttt{SimulationData.Dataset} object that can contain one element, use this format. Consider this form when writing scripts.

\section*{Output Arguments}

\textbf{element — Element}

\texttt{element | SimulationData.Dataset object | empty object}

The element that the \texttt{get} method finds.

- If \texttt{index} is the first argument after the data set, the method returns the element at the index.
- If \texttt{name} is the first argument after the data set:
  - If the method finds one element, it returns the element.
  - If the method finds more than one element, return a Dataset that contains the elements.
  - If the method does not find an element, it returns an empty object.

\section*{Examples}

\subsection*{Access Dataset Elements}

Access \texttt{Simulink.SimulationData.Dataset} elements in the top model of the \texttt{ex_bus_logging} model. The signal logging dataset is \texttt{topOut}.

\begin{verbatim}
open_system(docpath(fullfile(docroot,'toolbox','simulink','examples','ex_bus_logging')));
\end{verbatim}
open_system(fullfile(docroot,'toolbox','simulink',...'
examples','ex_mdlref_counter_bus'));
sim('ex_bus_logging')
topOut

```
topOut =

    Simulink.SimulationData.Dataset
    Package: Simulink.SimulationData

    Characteristics:
        Name: 'topOut'
        Total Elements: 4

    Elements:
        1: 'COUNTERBUS'
        2: 'OUTPUTBUS'
        3: 'INCREMENTBUS'
        4: 'inner_bus'

    -Use get or getElement to access elements by index, name or block path.
    -Use addElement or setElement to add or modify elements.
```

**Access Dataset Elements with Index**

Access the element at index if the first argument is a numeric value.

```
el = logsout.get(1);
```

**Access Dataset Elements with Characters**

Access the element whose name matches name.

```
el = logsout.get('name');
```
Access Dataset Elements with Cell Array

Return a dataset if the first argument is a cell array with a character vector as the first element.

\[
ds = \text{logsout.get}({'my\_name'});\]

Alternatives

You can use curly braces to streamline indexing syntax to get an element in a dataset, instead of using get or getElement. The index must be a scalar that is not greater than the number of elements in the variable. For example, get the second element of the logsout dataset.

\[
\text{logsout}\{2\}\]

Also, you can use the find method to get an element or collection of elements from a dataset.

See Also


Topics

“Migrate Scripts That Use Legacy ModelDataLogs API”
“Export Signal Data Using Signal Logging”
“Log Data Stores”
“Convert Logged Data to Dataset Format”
“Migrate Scripts That Use Legacy ModelDataLogs API”
Introduced in R2011a
getElementNames

**Class:** Simulink.SimulationData.Dataset  
**Package:** Simulink.SimulationData

Return names of all elements in dataset

**Syntax**

```
element_list = getElementNames(dataset)
```

**Description**

```
element_list = getElementNames(dataset) returns the names of all of the elements in the Simulink.SimulationData.Dataset object.
```

**Input Arguments**

- **dataset** — Data set  
  SimulationData.Dataset object  
  The data set from which to the element name.

**Output Arguments**

- **element_list** — Data set  
  cell array  
  Data set, returned as a cell array of the character vectors containing names of all of the elements of the dataset.

**Examples**
Return Names of Elements

Return the names of the elements for the `topOut` data set (the signal logging data).

```matlab
open_system(docpath(fullfile(docroot,'toolbox','simulink','...'
    'examples','ex_bus_logging')));
open_system(docpath(fullfile(docroot,'toolbox','simulink','...'
    'examples','ex_mdlref_counter_bus')));
sim('ex_bus_logging')
el_names = topOut.getElementNames

el_names =
    'COUNTERBUS'
    'OUTPUTBUS'
    'INCREMENTBUS'
    'inner_bus'
```

See Also

- `Simulink.SimulationData.BlockPath`
- `Simulink.SimulationData.DataStoreMemory`
- `Simulink.SimulationData.Dataset`
- `Simulink.SimulationData.Dataset.addElement`
- `Simulink.SimulationData.Dataset.concat`
- `Simulink.SimulationData.Dataset.find`
- `Simulink.SimulationData.Dataset.get`
- `Simulink.SimulationData.Dataset.numElements`
- `Simulink.SimulationData.Dataset.setElement`
- `Simulink.SimulationData.Signal`

Topics

- “Migrate Scripts That Use Legacy ModelDataLogs API”
- “Export Signal Data Using Signal Logging”
- “Log Data Stores”
- “Convert Logged Data to Dataset Format”
- “Migrate Scripts That Use Legacy ModelDataLogs API”

Introduced in R2011a
find

Class: Simulink.SimulationData.Dataset
Package: Simulink.SimulationData

Get element or collection of elements from dataset

Syntax

\[ \text{datasetOut, retIndex}\text{=}\text{find(datasetIn, Name, Value, }\ldots) \]
\[ \text{datasetOut, retIndex}\text{=}\text{find(datasetIn, Name, Value, '-logicaloperator', }\ldots\text{Name, Value, }\ldots) \]
\[ \text{datasetOut, retIndex}\text{=}\text{find(datasetIn, '-regexp', Name, Value, }\ldots) \]

Description

\[ \text{datasetOut, retIndex}\text{=}\text{find(datasetIn, Name, Value, }\ldots) \text{ returns a Simulink.SimulationData.Dataset object and indices of the elements whose property values match the specified property names and values. Specify optional comma-separated pairs of Name, Value properties. Name is the property name and Value is the corresponding value. Name must appear inside single quotes (' '). You can specify several name-value pair properties in any order as Name1, Value1, ..., NameN, ValueN.} \]
\[ \text{datasetOut, retIndex}\text{=}\text{find(datasetIn, Name, Value, '-logicaloperator', }\ldots\text{Name, Value, }\ldots) \text{ applies the logical operator to the matching property value. You can combine multiple logical operators. Logical operator can be one of:} \]
- or
- and

If you do not specify an operation, the method assumes -and.

\[ \text{datasetOut, retIndex}\text{=}\text{find(datasetIn, '-regexp', Name, Value, }\ldots \text{ matches elements using regular expressions as if the value of the property is passed to the regexp function as:} \]
regexp(element.Name,Value)

The method applies regular expression matching to the name-value pairs that appear after -regexp. If there is no -regexp, the method matches elements as if the value of the property is passed as:

isEqual(element.Name,Value)

For more information on -regexp, see “-regexp With Multiple Block Paths” on page 5-1251.

-regexp With Multiple Block Paths

-regexp works with properties of type char. To specify multiple block paths, you can use Simulink.SimulationData.BlockPath and Simulink.BlockPath. For example, when a signal is logged in a referenced model, you can use Simulink.SimulationData.BlockPath to specify multiple block paths.

The method returns elements that contain a BlockPath property where one or more of the individual block paths match the specified Value path when you use:

- -regexp with the BlockPath Name property.
- Value as a character vector or scalar object of type Simulink.SimulationData.BlockPath with one block path

Input Arguments

datasetIn — SimulationData.Dataset
SimulationData.Dataset object

SimulationData.Dataset object in which to search for matching elements.

Name — Name of property
character vector

Name of property to find in the element.

Value — Value of property
character vector | double | Simulink.SimulationData.BlockPath

Value of property to find in the element.
Output Arguments

datasetOut — SimulationData.Dataset data set
SimulationData.Dataset

SimulationData.Dataset object that contains the elements that match the specified criteria. If there is no matching SimulationData.Dataset object, the returned SimulationData.Dataset object contains no elements.

retIndex — Indices
vector

Indices of the elements datasetIn that match the specified criteria.

Examples

Find Block Path

Find a specific block path (specified by character vector) and port index.

dsOut = find(dsIn, 'BlockPath', 'vdp/x1', 'PortIndex', 1)

Find Elements

Find elements that have either name or propagated name as InValve.

dsOut = find(dsIn, 'Name', 'InValve', '-or', 'PropagatedName', 'InValve')
dsOut = find(dsIn, '-regex','Name', 'In*', '-or', ...
        '-regex','PropagatedName', 'In*')

Find and Change Element

Find and replace all elements containing specified_name with a new_name.

[dsOut,idxInDs] = find(ds, 'specified_name');
for idx=1:length(idxInDs)
    % process each element
elm = get(dsOut, idx);
elm.Name = 'New Name'
dsIn = setElement(dsIn, idxInDs(idx), elm);
end

**Find Signals in subSys Using -regexp**

Find all signals logged in a subSys using -regexp.

dsOut = find(dsIn, '-regexp', 'BlockPath', 'mdl/subSys/.*')

**Find Signals in Referenced Model**

Find all signals logged in the Model block.

dsOut = find(dsIn, '-regexp', 'BlockPath', 'refmdl/ModelBlk')

**Alternative**

You can use curly braces to streamline indexing syntax to get an element in a dataset, instead of using find. The index must be a scalar that is not greater than the number of elements in the variable. For example, get the second element of the logsout dataset.

logsout{2}

Also, you can use the get method to get an element or collection of elements from a dataset.

**See Also**

Simulink.SimulationData.BlockPath |
Simulink.SimulationData.DataStoreMemory |
Simulink.SimulationData.Dataset |
Simulink.SimulationData.Dataset.addElement |
Simulink.SimulationData.Dataset.concat |
Simulink.SimulationData.Dataset.get |
Simulink.SimulationData.Dataset getElementNames |
Simulink.SimulationData.Dataset.numElements |
Simulink.SimulationData.Dataset.setElement |
Simulink.SimulationData.DatasetRef.getDatasetVariableNames |
Simulink.SimulationData.Signal | findobj | regexp

**Topics**

“Migrate Scripts That Use Legacy ModelDataLogs API”
“Export Signal Data Using Signal Logging”
“Log Data Stores”
“Convert Logged Data to Dataset Format”
“Migrate Scripts That Use Legacy ModelDataLogs API”
“Load Big Data for Simulations”

**Introduced in R2015b**
numElements

**Class:** Simulink.SimulationData.Dataset  
**Package:** Simulink.SimulationData

Get number of elements in data set

**Syntax**

```matlab
length = numElements(dataset)
```

**Description**

`length = numElements(dataset)` gets the number of elements in the top-level dataset. To get the number of elements of a nested data set, use `numElements` with the nested data set.

**Input Arguments**

- **dataset** — Data set  
  SimulationData.Dataset object

  The data set from which to get the number of elements.

**Output Arguments**

- **length** — Number of elements  
  double

  Number of elements, returned as a double.

**Examples**
Get Number of Elements

Get the number of elements in the signal logging data set for the `ex_bus_logging`.

```
length = topOut.numElements()
```

See Also

- `Simulink.SimulationData.BlockPath`
- `Simulink.SimulationData.DataStoreMemory`
- `Simulink.SimulationData.Dataset`
- `Simulink.SimulationData.Dataset.addElement`
- `Simulink.SimulationData.Dataset.concat`
- `Simulink.SimulationData.Dataset.find`
- `Simulink.SimulationData.Dataset.get`
- `Simulink.SimulationData.Dataset.getElementNames`
- `Simulink.SimulationData.Dataset.setElement`
- `Simulink.SimulationData.Signal`

Topics

- “Migrate Scripts That Use Legacy ModelDataLogs API”
- “Export Signal Data Using Signal Logging”
- “Log Data Stores”
- “Convert Logged Data to Dataset Format”
- “Migrate Scripts That Use Legacy ModelDataLogs API”

Introduced in R2011a
plot

**Class:** Simulink.SimulationData.Dataset

**Package:** Simulink.SimulationData

Plot data in the Simulation Data Inspector

**Syntax**

plot(ds)
plot(ds,viewer)
runObj = plot(ds)

**Description**

**plot(ds)** plots the data in the Simulink.SimulationData.Dataset or Simulink.SimulationData.DatasetRef in the **Simulation Data Inspector**. When there are eight signals or fewer in the Dataset or DatasetRef object, the Simulation Data Inspector plot layout changes to 1-by-n, where n is equal to the number of signals, and plots one signal on each subplot. When the Dataset or DatasetRef object contains more than eight signals, the Simulation Data Inspector plot layout changes to 1-by-1 and plots the first signal.

**plot(ds,viewer)** displays the plot in the Simulation Data Inspector or Signal Preview window, according to the value specified for viewer.

**runObj = plot(ds)** plots the data in the Dataset object and returns the corresponding Simulink.sdi.Run object.

**Input Arguments**

ds — Dataset or DatasetRef object with data to plot
Simulink.SimulationData.Dataset object | Simulink.SimulationData.DatasetRef object

Dataset or DatasetRef object that contains the data to plot.
viewer — Specify viewer to use to plot data
'datainspector' (default) | 'preview'

Specify the viewer used to plot the data. By default, the plot function plots data in the Simulation Data Inspector. When you specify 'preview', the data displays in a read-only plot in the Signal Preview window.

Output Arguments

runObj — Simulink.sdi.Run object corresponding to the plotted data
Simulink.sdi.Run object

Simulink.sdi.Run object corresponding to the plotted data. When you view data in a Signal Preview window, the plot function returns empty ([ ]).

Examples

Plot a Data Set

Create a timeseries object ts and add elements to plot in the Simulation Data Inspector:

```matlab
% Create a timeseries object
ts = timeseries([0;20],[0;10]);
% Create a SimulationData.Dataset
ds = Simulink.SimulationData.Dataset();
% Place timeseries object in dataset
ds = ds.addElement(ts,'ts');
% Plot the element
plot(ds);
```

Compatibility Considerations

'sdi' input is not recommended

Not recommended starting in R2019b
Starting in R2019b, specify the viewer argument for the plot function as 'datainspector' to view plotted data in the Simulation Data Inspector.

**See Also**
Simulink.SimulationData.Dataset |
Simulink.SimulationData.Dataset.addElement

**Topics**
“View and Inspect Signal Data”
“Migrate Scripts That Use Legacy ModelDataLogs API”
“Export Signal Data Using Signal Logging”
“Log Data Stores”
“Convert Logged Data to Dataset Format”
“Migrate Scripts That Use Legacy ModelDataLogs API”
“Load Big Data for Simulations”

**Introduced in R2016b**
setElement

**Class:** Simulink.SimulationData.Dataset  
**Package:** Simulink.SimulationData

Change element stored at specified index

**Syntax**

```matlab
dataset = setElement(dataset,index,... element)
dataset = setElement(index,element, name)
```

**Description**

```matlab
dataset = setElement(dataset,index,... element) changes the element stored at the specified index, for an existing index. If index is one greater than the number of elements in the data set, the function adds the element at the end of the data set.

dataset = setElement(index,element, name) changes the element stored at the specified index and gives it the name that you specify. You can use name to identify an element that does not have a name. If the signal already has a name, the element instead uses the name you specify by using the name argument.
```

**Input Arguments**

```matlab
dataset — Data set  
SimulationData.Dataset object

The data set for which to set the element.

index — Index
scalar

Index for the added element, specified as a scalar numeric value. The value must be between 1 and the number of elements plus 1.
```
element — Element to replace existing element
Simulink.SimulationData.Signal object |
Simulink.SimulationData.DataStoreMemory object

Element to replace existing element or to add to the data set, specified as a
Simulink.SimulationData.Signal object or
Simulink.SimulationData.DataStoreMemory object.

name — Element name
character vector

Element name, returned as a character vector.

Output Arguments

dataset — Data set
character vector

Data set in which you change or add an element, specified as a character vector.

Examples

Set Element Name

Set element name.

ds = Simulink.SimulationData.Dataset
element1 = Simulink.SimulationData.Signal
element1.Name = 'A'
ds = ds.addElement(element1)
element2 = Simulink.SimulationData.Signal
element2.Name = 'B'
elementNew = Simulink.SimulationData.Signal
ds = ds.setElement(2,elementNew,'B1')
ds

ds =

Simulink.SimulationData.Dataset
Package: Simulink.SimulationData

Characteristics:
  Name: 'topOut'
  Total Elements: 2

Elements:
  1: 'A'
  2: 'B1'

Use getElement to access elements by index, name or block path.

Methods, Superclasses

Alternative

You can use curly braces to streamline indexing syntax to change an element in a dataset, instead of using setElement. The index must be a scalar that is not greater than the number of elements in the variable. For example, change the name of second element of the logsout dataset.

logsout{2}.Name = 'secondSignal'

See Also


Topics

“Migrate Scripts That Use Legacy ModelDataLogs API”
“Export Signal Data Using Signal Logging”
“Log Data Stores”
“Migrate Scripts That Use Legacy ModelDataLogs API”

**Introduced in R2011a**
**coder.BuildConfig class**

**Package:** coder

Build context during code generation

**Description**

The code generator creates an object of this class to facilitate access to the build context. The build context encapsulates the settings used by the code generator including:

- Target language
- Code generation target
- Target hardware
- Build toolchain

Use `coder.BuildConfig` methods in the methods that you write for the `coder.ExternalDependency` class.

**Construction**

The code generator creates objects of this class.
Methods

getHardwareImplementation  Get handle of copy of hardware implementation object
getStdLibInfo            Get standard library information
getTargetLang            Get target code generation language
getToolchainInfo         Returns handle of copy of toolchain information object
isCodeGenTarget          Determine if build configuration represents specified target
isMatlabHostTarget      Determine if hardware implementation object target is MATLAB host computer

Copy Semantics

Value. To learn how value classes affect copy operations, see Copying Objects (MATLAB).

Examples

Use `coder.BuildConfig` methods to access the build context in `coder.ExternalDependency` methods

This example shows how to use `coder.BuildConfig` methods to access the build context in `coder.ExternalDependency` methods. In this example, you use:

- `coder.BuildConfig.isMatlabHostTarget` to verify that the code generation target is the MATLAB host. If the host is not MATLAB report an error.
- `coder.BuildConfig.getStdLibInfo` to get the link-time and run-time library file extensions. Use this information to update the build information.

Write a class definition file for an external library that contains the function `adder`.

%================================================================
% This class abstracts the API to an external Adder library.
% It implements static methods for updating the build information
% at compile time and build time.
%================================================================
classdef AdderAPI < coder.ExternalDependency
    %#codegen

    methods (Static)

    function bName = getDescriptiveName(~)
        bName = 'AdderAPI';
    end

    function tf = isSupportedContext(buildContext)
        if buildContext.isMatlabHostTarget()
            tf = true;
        else
            error('adder library not available for this target');
        end
    end

    function updateBuildInfo(buildInfo, buildContext)
        % Get file extensions for the current platform
        [~, linkLibExt, execLibExt, ~] = buildContext.getStdLibInfo();

        % Add file paths
        hdrFilePath = fullfile(pwd, 'codegen', 'dll', 'adder');
        buildInfo.addIncludePaths(hdrFilePath);

        % Link files
        linkFiles = strcat('adder', linkLibExt);
        linkPath = hdrFilePath;
        linkPriority = 'i';
        linkPrecompiled = true;
        linkLinkOnly = true;
        group = '';
        buildInfo.addLinkObjects(linkFiles, linkPath, ...
            linkPriority, linkPrecompiled, linkLinkOnly, group);

        % Non-build files for packaging
        nbFiles = 'adder';
        nbFiles = strcat(nbFiles, execLibExt);
        buildInfo.addNonBuildFiles(nbFiles,'','');
    end

    %API for library function 'adder'
    function c = adder(a, b)
if coder.target('MATLAB')
    % running in MATLAB, use built-in addition
    c = a + b;
else
    % Add the required include statements to the generated function code
    coder.cinclude('adder.h');
    coder.cinclude('adder_initialize.h');
    coder.cinclude('adder_terminate.h');
    c = 0;
    % Because MATLAB Coder generated adder, use the
    % housekeeping functions before and after calling
    % adder with coder.ceval.
    coder.ceval('adder_initialize');
    c = coder.ceval('adder', a, b);
    coder.ceval('adder_terminate');
end
end
end

See Also
coder.ExternalDependency | coder.HardwareImplementation | coder.target

Topics
“Develop Interface for External C/C++ Code” (MATLAB Coder)
“Build Process Customization” (MATLAB Coder)

Introduced in R2013b
**coder.ExternalDependency class**

**Package:** coder

Interface to external code

**Description**

coder.ExternalDependency is an abstract class for developing an interface between external code and MATLAB code intended for code generation. You can define classes that derive from coder.ExternalDependency to encapsulate the interface to external libraries, object files, and C/C++ source code. This encapsulation allows you to separate the details of the interface from your MATLAB code.

To define a class derived from coder.ExternalDependency, create a subclass. For example:

```
classdef myClass < coder.ExternalDependency
```

You must define all of the methods listed in “Methods” on page 5-1269. These methods are static and are not compiled. The code generator invokes these methods in MATLAB after code generation is complete to configure the build for the generated code. The RTW.BuildInfo and coder.BuildConfig objects that describe the build information and build context are automatically created during the build process. The updateBuildInfo method provides access to these objects. For more information on build information customization, see “Build Process Customization” (MATLAB Coder).

You also define methods that call the external code. These methods are compiled. For each external function that you want to call, write a method to define the programming interface to the function. In the method, use coder.ceval to call the external function.
Methods

getDescriptiveName  Return descriptive name for external dependency
isSupportedContext  Determine if build context supports external dependency
updateBuildInfo    Update build information

Examples

Encapsulate the Interface to an External C Dynamic Library

This example shows how to encapsulate the interface to an external C dynamic linked library using coder.ExternalDependency.

Write a function adder that returns the sum of its inputs.

```matlab
function c = adder(a,b)
    c = a + b;
end
```

Generate a library that contains adder.

codegen('adder','-args',{-2,5},' -config: dll',' -report')

Write the class definition file AdderAPI.m to encapsulate the library interface.

```matlab
%================================================================
% This class abstracts the API to an external Adder library.
% It implements static methods for updating the build information
% at compile time and build time.
%================================================================

classdef AdderAPI < coder.ExternalDependency
    %#codegen

    methods (Static)
        function bName = getDescriptiveName(~)
            bName = 'AdderAPI';
        end
    end
```
function tf = isSupportedContext(buildContext)
    if buildContext.isMatlabHostTarget()
        tf = true;
    else
        error('adder library not available for this target');
    end
end

function updateBuildInfo(buildInfo, buildContext)
    % Get file extensions for the current platform
    [~, linkLibExt, execLibExt, ~] = buildContext.getStdLibInfo();
    % Add file paths
    hdrFilePath = fullfile(pwd, 'codegen', 'dll', 'adder');
    buildInfo.addIncludePaths(hdrFilePath);
    % Link files
    linkFiles = strcat('adder', linkLibExt);
    linkPath = hdrFilePath;
    linkPriority = '';
    linkPrecompiled = true;
    linkLinkOnly = true;
    group = '';
    buildInfo.addLinkObjects(linkFiles, linkPath, ...
                              linkPriority, linkPrecompiled, linkLinkOnly, group);
    % Non-build files for packaging
    nbFiles = 'adder';
    nbFiles = strcat(nbFiles, execLibExt);
    buildInfo.addNonBuildFiles(nbFiles,'','');
end

% API for library function 'adder'
function c = adder(a, b)
    if coder.target('MATLAB')
        % running in MATLAB, use built-in addition
        c = a + b;
    else
        % Add the required include statements to the generated function code
        coder.cinclude('adder.h');
        coder.cinclude('adder_initialize.h');
        coder.cinclude('adder_terminate.h');
        c = 0;
    end
end
% Because MATLAB Coder generated adder, use the
% housekeeping functions before and after calling
% adder with coder.ceval.

coder.ceval('adder_initialize');
c = coder.ceval('adder', a, b);
coder.ceval('adder_terminate');

Write a function adder_main that calls the external library function adder.

function y = adder_main(x1, x2)
    %#codegen
    y = AdderAPI.adder(x1, x2);
end

Generate a MEX function for adder_main. The MEX Function exercises the
coder.ExternalDependency methods.

codegen('adder_main','-args',{7,9},'-report')

Copy the library to the current folder using the file extension for your platform. For
Windows, use:
copyfile(fullfile(pwd,'codegen','dll','adder','adder.dll'));

For Linux, use:
copyfile(fullfile(pwd,'codegen','dll','adder','adder.so'));

Run the MEX function and verify the result.
adder_main_mex(2,3)

See Also
 coder.BuildConfig | coder.ceval | coder.cinclude | coder.updateBuildInfo

Topics
“Develop Interface for External C/C++ Code” (MATLAB Coder)
“Build Process Customization” (MATLAB Coder)
“Integrate External/Custom Code” (MATLAB Coder)

**Introduced in R2013b**
**TimeScopeConfiguration**

Control Scope block appearance and behavior

**Description**

Scope configuration properties control the appearance and behavior of a scope block. Create a scope configuration object with `get_param`, and then change property values using the object with dot notation.

**Creation**

`myScopeConfiguration = get_param(gcbh,'ScopeConfiguration')` creates a scope configuration object for the selected scope block.

**Properties**

**Name — Title on a scope window**

block name (default) | character vector | string scalar

Title on a scope window, specified as a character vector or string scalar.

**NumInputPorts — Number of input ports**

'1' (default) | character vector

Number of input ports on a scope block, specified as a character vector or string scalar. The maximum number of input ports is 96.

**UI Use**

Select **File > Number of Input Ports**.

**ActiveDisplay — Display for setting display-specific properties**

'1' (default) | character vector
Display for setting display-specific properties, specified as a character vector or string. The number of a display corresponds to its column-wise placement index. For multi-column layouts, the displays are numbered down and then across.

**Dependency**

Setting this property selects the display for setting the properties `ShowGrid`, `ShowLegend`, `Title`, `PlotAsMagnitudePhase`, `YLabel`, and `YLimits`.

**UI Use**

Open the **Configuration Properties**. On the **Display** tab, set **Active display**.

**AxesScaling — How to scale y-axes**

'Manual' (default) | 'Auto' | 'Updates'

How to scale y-axes, specified as one of these values:

- 'Manual' — Manually scale y-axes with the **Scale Y-axis Limits** button.
- 'Auto' — Scale y-axes during and after simulation.
- 'Updates' — Scale y-axes after specified number of block updates (time intervals).

**Dependency**

If this property is set to 'Updates', also specify the property `AxesScalingNumUpdates`

**UI Use**

Open the **Configuration Properties**. On the **Main** tab, set **Axes scaling**.

**AxesScalingNumUpdates — Number of updates before scaling y-axes**

'10' (default) | character vector | string scalar

Number of updates before scaling y-axes, specified as a character vector or string scalar.

**Dependency**

Activate this property by setting `AxesScaling` to 'Updates'.

**UI Use**

Open the **Configuration Properties**. On the **Main** tab, set **Number of updates**.

**DataLogging — Save scope data**

false (default) | true
Set this property to `true` to save scope data to a variable in the MATLAB workspace.

This property does not apply to floating scopes and scope viewers.

**Dependency**

If this property is set to `true`, you must also specify the properties `DataLoggingVariableName` and `DataLoggingSaveFormat`.

**UI Use**

Open the **Configuration Properties**. On the **Logging** tab, set **Log data to workspace**.

`DataLoggingVariableName — Variable name for saving scope data`

'ScopeData' (default) | character vector | string scalar

Variable name for saving scope data in the MATLAB workspace, specified as a character vector or string scalar. This property does not apply to floating scopes and scope viewers.

**Dependency**

Activate this property by setting `DataLogging` to `true`.

**UI Use**

Open the **Configuration Properties**. On the **Logging** tab, set **Variable name**.

`DataLoggingSaveFormat — Variable format for saving scope data`

'Dataset' (default) | 'Structure With Time' | 'Structure' | 'Array'

Variable format for saving scope data to the MATLAB workspace, specified as one of these values:

- 'Dataset' — Save data as a dataset object. This format does not support variable-size data, MAT-file logging, or external mode archiving. See `Simulink.SimulationData.Dataset`.
- 'StructureWithTime' — Save data as a structure with associated time information. This format does not support single- or multiport frame-based data, or multirate data.
- 'Structure' — Save data as a structure. This format does not support multirate data.
- 'Array' — Save data as an array with associated time information. This format does not support multiport sample-based data, single- or multiport frame-based data, variable-size data, or multirate data.
This property does not apply to floating scopes and scope viewers.

**Dependency**

Activate this property by setting `DataLogging` to `true`.

**UI Use**

Open the **Configuration Properties**. On the **Logging** tab, set **Save format**.

**DataLoggingLimitDataPoints — Limit buffered data**

`false` (default) | `true`

Set to `true` to limit buffered data before plotting and saving data.

For simulations with **Stop time** set to `inf`, always set this parameter to `true`.

**Dependency**

If this property is set to `true`, also specify the number of data values to plot and save with the property `DataLoggingMaxPoints`.

**UI Use**

Open the **Configuration Properties**. On the **Logging** tab, set **Limit data points to last**.

**DataLoggingMaxPoints — Maximum number of data values**

`'5000'` (default) | character vector | string scalar

Maximum number of data values to plot and save, specified as a character vector or string scalar. The data values that are plotted and saved are from the end of a simulation. For example, setting this property to 100 saves the last 100 data points.

**Dependency**

Activate this property by setting `DataLoggingLimitDataPoints` to `true`. Specifying this property limits the data values a scope plots and the data values saved in the MATLAB variable specified in `DataLoggingVariableName`.

**UI Use**

Open the **Configuration Properties**. On the **Logging** tab, set the text box to the right of **Limit data points to last**.
DataLoggingDecimateData — Reduce scope data
false (default) | true

Set this property to true to reduce scope data before plotting and saving.

Dependency

If this property is set to true, you must also specify the DataLoggingDecimation property.

UI Use

Open the Configuration Properties. On the Logging tab, set Decimation.

DataLoggingDecimation — Decimation factor
'1' (default) | character vector | string scalar

Decimation factor applied to the signal data before plotting and saving, specified as a character vector or string scalar. The scope buffers every $N^{th}$ data point, where $N$ is the decimation factor you specify. A value of 1 buffers all data values.

Dependency

Activate this property by setting DataLoggingDecimateData to true.

UI Use

Open the Configuration Properties. On the Logging tab, set the text box to the right of Decimation.

FrameBasedProcessing — Frame-based processing of signals
false (default for Time Scope block) | true (default for Scope block)

Set this property to true to process signals as frame-based.

• false — Process signal values in a channel at each time interval (sample based).
• true — Process signal values in a channel as a group of values from multiple time intervals (frame based). Frame-based processing is available only with discrete input signals.

UI Use

Open the Configuration Properties. On the Main tab, set Input processing.
**LayoutDimensions — Number of display rows and columns**

[1 1](default) | [numberOfRows numberOfColumns]

Number of display rows and columns, specified with as a two-element vector. The maximum layout dimension is 16-by-16.

- If the number of displays is equal to the number of ports, signals from each port appear on separate displays.
- If the number of displays is less than the number of ports, signals from additional ports appear on the last y-axis.

**UI Use**

Open the **Configuration Properties**. On the **Main** tab, select the **Layout** button.

**MaximizeAxes — Maximize size of signal plots**

'Auto' (default) | 'On' | 'Off'

Specify whether or not to maximize the size of signal plots:

- 'Auto' — If Title and YLabel are not specified, maximize all plots.
- 'On' — Maximize all plots. Values in Title and YLabel are hidden.
- 'Off' — Do not maximize plots.

Each of the plots expands to fit the full display. Maximizing the size of signal plots removes the background area around the plots.

**UI Use**

Open the **Configuration Properties**. On the **Main** tab, set **Maximize axes**.

**MinimizeControls — Hide menu and toolbar**

false (default) | true

Set this property to true to hide the menu and toolbar.

If you dock the scope, this property is inactive.

**OpenAtSimulationStart — Open scope when starting simulation**

true (default for Time Scope) | false (default for Scope)

Set this property to true to open the scope when the simulation starts.
UI Use

Select File > Open at Start of Simulation

PlotAsMagnitudePhase — Magnitude and phase plots
false (default) | true

Specify whether or not to display the magnitude and phase plots:

- false — Display signal plot.
  
  If the signal is complex, plot the real and imaginary parts on the same y-axis (display).
- true — Display magnitude and phase plots.
  
  If the signal is real, plot the absolute value of the signal for the magnitude. The phase is 0 degrees for positive values and 180 degrees for negative values.

Dependency

Set the ActiveDisplay property before setting this property.

UI Use

Open the Configuration Properties. On the Display tab, set Plot signals as magnitude and phase.

Position — Size and location of the scope
[left bottom width height]

Size and location of scope window, specified as a four-element vector consisting of the left, bottom, width, and height positions, in pixels.

By default, a scope window appears in the center of your screen with a width of 560 pixels and height of 420 pixels.

ShowGrid — Vertical and horizontal grid lines
true (default) | false

Set this property to true to display vertical and horizontal grid lines.

Dependency

Set the ActiveDisplay property before setting this property.
UI Use

SampleTime — Time interval
'-1' (default) | character vector | string scalar
Time interval between Scope block updates during a simulation, specified as a character vector or string scalar. This property does not apply to floating scopes and scope viewers.

UI Use
Open the Configuration Properties. On the Main tab, set Sample Time.

ShowLegend — Signal legend
false (default) | true
Set this property to true to display the legend.
Names listed in the legend are the signal names from the model. For signals with multiple channels, a channel index is appended after the signal name. See the Scope block reference for an example.

Dependency
Set the ActiveDisplay property before setting this property.

UI Use
Open the Configuration Properties. On the Display tab, set Show legend.

ShowTimeAxisLabel — Display or hide x-axis labels
true (default for Time Scope block) | false (default for Scope block)
Set this property to true to display the x-axis labels.

Dependency
Set the ActiveDisplay property before setting this property.
If this property is set to true, also set TimeAxisLabels. If TimeAxisLabels is set to 'None', this property is inactive.

UI Use
Open the Configuration Properties. On the Time tab, set Show time-axis label.
**TimeAxisLabels — How x-axis labels display**

'All' (default for Time Scope block) | 'Bottom' (default for Scope block) | 'None'

How x-axis labels display, specified as one of these values:

- 'All' — Display x-axis labels on all displays.
- 'Bottom' — Display x-axis labels only on the bottom display.
- 'None' — Do not display labels and deactivate `ShowTimeAxisLabel` property.

**Dependency**

Set the `ActiveDisplay` property before specifying this property.

Set `ShowTimeAxisLabel` to `true` and set `Maximize axes` to 'Off'.

**UI Use**

Open the **Configuration Properties**. On the **Time** tab, set **Time-axis labels**.

**TimeDisplayOffset — x-axis range offset**

'0' (default) | character vector | string scalar

x-axis range offset number, specified as a character vector or string scalar. For input signals with multiple channels, enter a scalar or vector of offsets.

- Scalar — Offset all channels of an input signal by the same value.
- Vector — Independently offset the channels.

**UI Use**

Open the **Configuration Properties**. On the **Time** tab, set **Time display offset**.

**TimeSpan — Length of x-axis range to display**

'0' (default) | character vector | string scalar | 'Auto'

Length of x-axis range to display, specified as one of these values:

- Positive real number — Any value less than the total simulation time specified as a character vector or string scalar.
- 'Auto' — Difference between the simulation start and stop times.
The block calculates the beginning and end times of the x-axis range using the `TimeDisplayOffset` and `TimeSpan` properties. For example, if you set `TimeDisplay` to 10 and the `TimeSpan` to 20, the scope sets the x-axis range from 10 to 30.

**UI Use**

Open the **Configuration Properties**. On the **Time** tab, set **Time span**.

**TimeSpanOverrunAction — How to display data**

'Wrap' (default) | 'Scroll'

How to display data beyond the visible x-axis range, specified as one of these values:

- 'Wrap' — Draw a full screen of data from left to right, clear the screen, and then restart drawing of data.
- 'Scroll' — Move data to the left as new data is drawn on the right. This mode is graphically intensive and can affect run-time performance.

You can see the effects of this option only when plotting is slow with large models or small step sizes.

**UI Use**

Open the **Configuration Properties**. On the **Time** tab, set **Time span overrun action**.

**TimeUnits — Units to display on the x-axis**

'Metric' (default for Time Scope block) | 'None' (default for Scope block) | 'Seconds'

Units to display on the x-axis, specified as one of these values:

- 'Metric' — Display time units based on the length of the `TimeSpan` property.
- 'None' — Display Time on the x-axis.
- 'Seconds' — Display Time (seconds) on the x-axis.

**UI Use**

Open the **Configuration Properties**. On the **Time** tab, set **Time units**.

**Title — Title for display**

'%'<SignalLabel>' (default) | character vector | string scalar

Title for a display, specified as a character vector or string scalar. The default value '%<SignalLabel>' uses the input signal name for the title.
Dependency

Set the ActiveDisplay property before setting this property.

UI Use

Open the Configuration Properties. On the Display tab, set Title.

Visible — Visibility of scope window
true (default) | false

Set this property to true to make the scope window visible.

YLabel — Y-axis label
'' (default) | character vector | string scalar

y-axis label for active display, specified as a character vector or string scalar.

Dependency

Set the ActiveDisplay property before setting this property.

If PlotAsMagnitudePhase is true, the value of YLabel is hidden and plots are labeled Magnitude and Phase.

UI Use


YLimits — Minimum and maximum values of y-axis
[-10 10] (default) | [ymin ymax]

Minimum and maximum values of y-axis, specified as a two-element numeric vector.

Dependency

Set the ActiveDisplay property before setting this property.

When PlotAsMagnitudePhase is true, this property specifies the y-axis limits for the magnitude plot. The y-axis limits of the phase plot are always [-180 180].

UI Use

Open the Configuration Properties. On the Display tab, set Y-limits (Minimum) and Y-limits (Maximum).
Examples

Create Scope Configuration Object

This example creates a scope configuration object using the 'vdp' model which models the van der Pol equation.

```matlab
open_system('vdp')
myScopeConfiguration = get_param('vdp/Scope','ScopeConfiguration');
myScopeConfiguration.NumInputPorts = '2';
```

See Also
Floating Scope | Scope | Time Scope

Topics
“Control Scope Blocks Programmatically”

Introduced in R2013a
Simulink.Simulation.Job class

Package: Simulink

Execute `batchsim` to create a `Simulink.Simulation.Job` object, `simJob`

Description

Execute `batchsim` command with a parallel pool to create a `Simulink.Simulation.Job` object. The `batchsim` command offloads multiple simulations to run in batch using the inputs specified with an array of `Simulink.SimulationInput` objects. You can use this object to monitor the status of a batch job, fetch outputs of a completed batch job, or cancel one or more jobs.

The `batchsim` command uses the Parallel Computing Toolbox™ license to run the simulations on compute cluster. If a parallel pool cannot be created `batchsim` runs the simulations in serial. In the absence of Parallel Computing Toolbox license, `batchsim` errors out.

Construction

`simJob = batchsim(in)` creates a `Simulink.Simulation.Job` object, `simJob`, while running multiple simulations in batches on a compute cluster using the inputs specified in the `Simulink.SimulationInput` object, `in`.

`createSimulationJob(batchJob)` creates a `Simulink.Simulation.Job` object from a `parallel.job` object, `batchJob`. Use the `createSimulationJob` command to retrieve the job object returned by the `batchsim` command.

Input Arguments

`in` — `Simulink.SimulationInput` object array

`object (default) | array`

A `Simulink.SimulationInput` object or an array of `Simulink.SimulationInput` objects used to run multiple simulations. Specify parameters and values of a model to run multiple simulations without making it dirty.
Example: in = Simulink.SimulationInput('vdp'), in(1:10) = Simulink.SimulationInput('vdp')

**Properties**

**AdditionalPaths — Folders to add to MATLAB search path of workers**
character vector | string | string array | array of character vectors

Specified folders to add to MATLAB search path of parallel workers.

This property is read-only.

**AttachedFiles — Files and folders sent to the workers**
character vector | string | string array | array of character vectors

Files and folders to send to the parallel workers.

This property is read-only.

**AutoAddClientPath — Whether user-added entries on client path are added to each worker path**
true (default) | false

Whether user-added entries on the client path are added to each parallel worker path at the start of the batch job, specified as true or false.

**CreateDateDateTime — Date and time of simJob creation**
datetime object

Date and time at which the batch job was created, specified as a datetime object.

This property is read-only.

**EnvironmentVariables — Environment variables sent to the workers**
character vector | string | string array | array of character vectors

Defines the names of environment variables that are copied from a client session to the parallel workers.

This property is read-only after job submission.
**FinishDateTime** — Date and time of simJob completion

datetime object

Date and time at when the batch job completes execution, specified as a datetime object.

This property is read-only.

**ID** — Numeric identifier of the simJob object

scalar integer

ID of the future object, specified as a scalar integer.

This property is read-only.

**Name** — simJob name

string | character vector

Name of the job object, specified as a string.

**Parent** — Cluster object containing simJob

parallel.cluster object


**StartDateTime** — Date and time when simJob starts running

datetime object

Date and time when the Simulink.Simulation.Job starts running, specified as a datetime object.

This property is read-only.

**State** — Current state of future object array

'pending' | 'queued' | 'running' | 'finished' | 'failed'

Current state of future object array, specified as 'pending', 'queued', 'running', 'finished', or 'failed'.

This property is read-only.

**SubmitDateTime** — Date and time of simJob submission

datetime object
Date and time when the Simulink.Simulation.Job is submitted, specified as a datetime object.

This property is read-only.

**Tag — Label associated with simJob**

string


**Type — Job Type**

independent | pool

Type of the Simulink.Simulation.Job object, specified as independent or pool.

**UserData — Data associated with simJob**

string

Stores any data associated with a job object. The data is stored in a client MATLAB session, and is not available on the workers.

**UserName — Name of the user who creates the simJob object**

string | character vector

Name of the user who creates the Simulink.Simulation.Job object.

## Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>cancel</td>
<td>Cancel a pending, queued, or running Simulink.Simulation.Job object</td>
</tr>
<tr>
<td>diary</td>
<td>Display or save Command Window text of batch job</td>
</tr>
<tr>
<td>fetchOutputs</td>
<td>Retrieve an array of Simulink.SimulationOutput objects from all simulations in Simulink.Simulation.Job</td>
</tr>
<tr>
<td>listAutoAttachedFiles</td>
<td>List of files automatically attached to job, task, or parallel pool</td>
</tr>
</tbody>
</table>
Method | Purpose
---|---
wait | Wait for Simulink.Simulation.Job object to change state

Examples

Run Parallel Simulations With `batchsim` to Create `Simulink.Simulation.Job`

This example shows how to run parallel simulations in batch using the `sldemo_househeat` model. `batchsim` offloads simulations to the compute cluster, enabling you to carry out other tasks while the batch job is processing, or close the client MATLAB and access the batch job later.

Observe the model behavior for different temperature set points.

1. Open the model.
   
   ```matlab
   open_system('sldemo_househeat');
   ```

2. Define a set of values for different temperatures.
   
   ```matlab
   setPointValues = 65:2:85;
   spv_Length = length(setPointValues);
   ```

3. Using the `setPointValues`, initialize an array of `Simulink.SimulationInput` objects.
   
   ```matlab
   in(1:spv_Length) = Simulink.SimulationInput('sldemo_househeat');
   for i = 1:spv_Length
       in(i) = in(i).setBlockParameter('sldemo_househeat/Set Point',
                                    'Value',num2str(setPointValues(i)));
   end
   ```

4. Specify the pool size of the number of workers to use. In addition to the number of workers used to run simulations in parallel, a head worker is required. In this case, assume that three workers are available to run a batch job for the parallel simulations. The job object returns useful metadata as shown. You can use the job ID to access the job object later from any machine. `NumWorkers` displays how many workers are running the simulations - the number of workers specified in the 'Pool' argument plus an additional head worker.
simJob = batchsim(in,'Pool',3)

    ID: 1
    Type: pool
    NumWorkers: 4
    Username: ######
    State: running
    SubmitDateTime: ##-###-#### ##:##:##
    StartDateTime:
    Running Duration: 0 days 0h 0m 0s

See Also

Functions
batchsim | cancel | diary | fetchOutputs | getSimulationJobs |
listAutoAttachedFiles | parcluster | parsim | wait

Classes

Topics
“Multiple Simulation Workflows”
“Job Monitor” (Parallel Computing Toolbox)
“Batch Processing” (Parallel Computing Toolbox)

Introduced in R2018b
cancel

Package: Simulink

Cancel a pending, queued, or running Simulink.Simulation.Job object

Syntax

cancel(simJob)
cancel(simJob,'Message')

Description

cancel(simJob) stops the Simulink.Simulation.Job object, simJob, that is currently in 'pending', 'queued', or 'running' state.

cancel(simJob,'Message') stops the Simulink.Simulation.Job object, simJob, that is in 'pending', 'queued', or 'running' state and displays a user-specified message.

The State property of Simulink.Simulation.Job object is set to finished, and other pending simulations are canceled. Canceling a job object disables you to fetch results from it. A canceled job object cannot be started again.

Note On canceling the job, the results of the completed simulations in the job are also lost.

Examples

Cancel a Batch Job

This example shows how to use the cancel method on a simJob object to stop simulations. The example runs several simulations of the vdp model, varying the value of the gain Mu.
1. Open the model and define a vector of Mu values.

```matlab
open_system('vdp');
mu_Values = [0.5:0.25:1000];
muVal_Length = length(mu_Values);
```


```matlab
in(1:muVal_Length) = Simulink.SimulationInput('vdp');
for i = 1:1:muVal_Length
    in(i) = in(i).setBlockParameter('vdp/Mu','Gain',num2str(mu_Values(i)));
end
```

3. Specify the pool size of the number of workers to use. In addition to the number of workers used to run simulations in parallel, a head worker is required. In this case, assume that three workers are available to run a batch job for the parallel simulations. The simulations are offloaded onto the default cluster profile.

```matlab
simJob = batchsim(in,'Pool',3);
```

4. Now, assume that you want to run simulations with different values of Mu and cancel the ongoing simulations.

```matlab
cancel(simJob)
```

### Input Arguments

- **simJob** — `Simulink.Simulation.Job` object
  
  
  Example: `simJob = batchsim(in,'Pool',6)`

### See Also

- **Functions**  
  
  - `batch`  
  - `batchsim`  
  - `diary`  
  - `fetchOutputs`  
  - `getSimulationJobs`  
  - `listAutoAttachedFiles`  
  - `parsim`  
  - `wait`
Classes
Simulink.Simulation.Job | Simulink.SimulationInput

Topics
“Multiple Simulation Workflows”
“Batch Processing” (Parallel Computing Toolbox)

Introduced in R2018b
**diary**

**Package:** Simulink

Display or save Command Window text of Simulink.Simulation.Job object

**Syntax**

```matlab
diary(simJob)
diary(simJob,'filename')
```

**Description**

diary(simJob) displays the Command Window output from the Simulink.Simulation.Job object, simJob, in the MATLAB Command Window. The Command Window output is captured only if the batchsim command includes the 'CaptureDiary' argument with a value of true.

diary(simJob,'filename') causes the Command Window output from the batch job to be appended to the specified file. Open the file, filename, with any text editor.

**Examples**

**Display the Diary of simJob**

This example uses sldemo_househeat model to show how to display the diary of the Simulink.Simulation.Job object, simJob. To create a simJob, you run parallel simulations using the batchsim command.

1. Open the model.

```matlab
open_system('sldemo_househeat');
```

2. Define a set of values for different temperatures.
setPointValues = 65:2:85;
spv_Length = length(setPointValues);


```matlab
in(1:spv_Length) = Simulink.SimulationInput('sldemo_househeat');
for i = 1:1:spv_Length
    in(i) = in(i).setBlockParameter('sldemo_househeat/Set Point', ...
                                          'Value',num2str(setPointValues(i)));
end
```

4. Specify the pool size of the number of workers to use. In addition to the number of workers used to run simulations in parallel, a head worker is required. In this case, assume that three workers are available to run a batch job for the parallel simulations. The job object returns useful metadata as shown. You can use the job ID to access the job object later from any machine. `NumWorkers` displays how many workers are running the simulations – the number of workers specified in the 'Pool' argument plus an additional head worker.

```matlab
simJob = batchsim(in,'Pool',3)
```

```
ID: 1
Type: pool
NumWorkers: 4
Username: ######
State: running
SubmitDateTime: ##-###-#### ##:##:##
StartDateTime:
Running Duration: 0 days 0h 0m 0s
```

5. Use the `diary` method of the `Simulink.Simulation.Job` object to display the output of the batch job in the MATLAB command window.

Note that the diary is not displayed here because this is an example model.

```matlab
diary(simJob)
```

**Input Arguments**

`simJob` — `Simulink.Simulation.Job` object


5-1295
Example: `simJob = batchsim(in,'Pool',4)`

'filename' — File to append with Command Window output text

String

Specify a file to append with Command Window output text from the `Simulink.Simulation.Job` object.

Example: `diary(simJob,'abc.txt')`

See Also

Functions
- `batchsim`
- `cancel`
- `fetchOutputs`
- `getSimulationJobs`
- `listAutoAttachedFiles`
- `parcluster`
- `parsim`
- `wait`

Classes
- `Simulink.Simulation.Job`
- `Simulink.SimulationInput`
- `parallel.Cluster`

Topics
- “Multiple Simulation Workflows”
- “Batch Processing” (Parallel Computing Toolbox)

Introduced in R2018b
fetchOutputs

Package: Simulink

Retrieve an array of Simulink.SimulationOutput objects from all simulations in Simulink.Simulation.Job

Syntax

out = fetchOutputs(simJob)

Description

out = fetchOutputs(simJob) returns an array of Simulink.SimulationOutput objects containing the results of the simulations in a batch job, simJob.

fetchOutputs reports an error if the job is not in the 'finished' state, or if one of its simulations encounters an error during execution. Use the wait method to wait for the job to complete before fetching outputs.

Examples

Fetch Outputs of the Batch Simulations

This example shows how to run parallel simulations in batch and fetch the resulting Simulink.SimulationOutput objects from the Simulink.Simulation.Job object. batchsim command offloads simulations to the compute cluster, enabling you to carry out other tasks while the batch job is processing, or close the client MATLAB and access the batch job later.

1. Open the model.

open_system('sldemo_househeat');

2. Define a set of values for different temperatures.
setPointValues = 65:2:85;
spv_Length = length(setPointValues);


in(1:spv_Length) = Simulink.SimulationInput('sldemo_househeat');
for i = 1:1:spv_Length
    in(i) = in(i).setBlockParameter('sldemo_househeat/Set Point', ...
        'Value',num2str(setPointValues(i)));
end

4. Specify the pool size of the number of workers to use. In addition to the number of workers used to run simulations in parallel, a head worker is required. In this case, assume that three workers are available to run a batch job for the parallel simulations. The job object returns useful metadata as shown. You can use the job ID to access the job object later from any machine. NumWorkers displays how many workers are running the simulations - the number of workers specified in the 'Pool' argument plus an additional head worker.

simJob = batchsim(in,'Pool',3)

    ID: 1
    Type: pool
    NumWorkers: 4
    Username: ######
    State: running
    SubmitDateTime: ##-###-#### ##:##:##
    StartDateTime: 
    Running Duration: 0 days 0h 0m 0s

5. Access the results of the batch job using the fetchOutputs method. fetchOutputs returns an array of Simulink.SimulationOutput objects. You can fetch outputs only once simJob is in finished state.

out = fetchOutputs(simJob)

1x11 Simulink.SimulationOutput array

**Input Arguments**

simJob — Simulink.Simulation.Job object
object
Example: simJob = batchsim(in,'Pool',4)

Output Arguments

out — Simulation object containing logged simulation results
object

Array of Simulink.SimulationOutput objects that contain all of the logged simulation results. The size of the array is equal to the size of the array of Simulink.SimulationInput objects passed to batchsim.

All simulation outputs (logged time, states, and signals) are returned in a single Simulink.SimulationOutput object. You define the model time, states, and output that are logged using the Data Import/Export pane of the Model Configuration Parameters dialog box. You can log signals using blocks such as the To Workspace and Scope blocks. The Signal & Scope Manager tool can directly log signals.

See Also

Functions
batchsim | cancel | diary | getSimulationJobs | listAutoAttachedFiles | parsim | wait

Classes

Topics
“Multiple Simulation Workflows”
“Batch Processing” (Parallel Computing Toolbox)

Introduced in R2018b
listAutoAttachedFiles

**Package:** Simulink

List of files automatically attached to the Simulink.Simulation.Job object or parallel pool

**Syntax**

listAutoAttachedFiles(simJob)

**Description**

listAutoAttachedFiles(simJob) performs a dependency analysis on the Simulink.Simulation.Job job object, simJob. Then it displays a list of the code files that are already attached or are going to be automatically attached to the job object, simJob.

**Examples**

**Run Parallel Simulations with batchsim and List Attached Files**

This example shows how to run parallel simulations in batch and list any attached files. batchsim offloads simulations to the compute cluster, enabling you to carry out other tasks while the batch job is processing, or close the client MATLAB and access the batch job later.

1. Open the model.

   open_system('sldemo_househeat');

2. Define a set of values for different temperatures.

   setPointValues = 65:2:85;
   spv_Length = length(setPointValues);
3. Using the `setPointValues`, initialize an array of `Simulink.SimulationInput` objects.

```matlab
in(1:spv_Length) = Simulink.SimulationInput('sldemo_househeat');
for i = 1:1:spv_Length
    in(i) = in(i).setBlockParameter('sldemo_househeat/Set Point', ...    
        'Value', num2str(setPointValues(i)));
end
```

4. Specify the pool size of the number of workers to use. In addition to the number of workers used to run simulations in parallel, a head worker is required. In this case, assume that three workers are available to run a batch job for the parallel simulations. The job object returns useful metadata as shown. You can use the job ID to access the job object later from any machine. `NumWorkers` displays how many workers are running the simulations – the number of workers specified in the 'Pool' argument plus an additional head worker.

```matlab
simJob = batchsim(in,'Pool',3)
```

```plaintext
ID: 1
Type: pool
NumWorkers: 4
Username: #####
State: running
SubmitDateTime: ##-###-#### ##:##:##
StartDateTime: 
Running Duration: 0 days 0h 0m 0s
```

If `AutoAttachFiles` property of `Simulink.Simulation.Job` is set to true in the cluster profile, then the job running on the cluster has the necessary code files automatically attached to it. Use the `listAutoAttachedFiles` method to view the attached files.

Note that the list of attached files is not displayed here because this is an example model.

```matlab
listAutoAttachedFiles(simJob)
```

### Input Arguments

- **`simJob` — Simulink.Simulation.Job object**
  

5-1301
Example: simJob = batchsim(in,'Pool',4)

See Also

Functions
batchsim | cancel | diary | fetchOutputs | getSimulationJobs | parsim | wait

Classes
Simulink.Simulation.Job | Simulink.SimulationInput

Topics
“Multiple Simulation Workflows”
“Batch Processing” (Parallel Computing Toolbox)

Introduced in R2018b
wait

**Package:** Simulink

Wait for Simulink.Simulation.Job object to change state

### Syntax

```matlab
wait(simJob)
wait(simJob,'stateOfJob')
wait(simJob,'stateOfJob',timeout)
```

### Description

`wait(simJob)` blocks execution in a client session until the job identified by the object `simJob` reaches the 'finished' state or fails. This occurs when all the simulations finish execution on the workers.

`wait(simJob,'stateOfJob')` blocks execution in the client session until the specified job object changes state to the value of 'state'. The valid states to wait for are 'queued', 'running', and 'finished'. If the object is currently or has already been in the specified state, `wait` is not performed and execution returns immediately. For example, if you execute `wait(simJob,'queued')` for a job already in the 'finished' state, the call returns immediately.

`wait(simJob,'stateOfJob',timeout)` blocks execution until either the job reaches the specified 'state', or timeout seconds elapse, whichever happens first.

### Examples

**Wait for Simulations in simJob to Finish**

This example shows uses the `sldemo_househeat` model to show how to wait for the batch simulations to finish.
1. Open the model.

```matlab
open_system('sldemo_househeat');
```

2. Define a set of values for different temperatures.

```matlab
setPointValues = 65:2:85;
spv_Length = length(setPointValues);
```

3. Using the `setPointValues`, initialize an array of `Simulink.SimulationInput` objects.

```matlab
in(1:spv_Length) = Simulink.SimulationInput('sldemo_househeat');
for i = 1:1:spv_Length
    in(i) = in(i).setBlockParameter('sldemo_househeat/Set Point', ...
        'Value',num2str(setPointValues(i)));
end
```

4. Specify the pool size of the number of workers to use. In addition to the number of workers used to run simulations in parallel, a head worker is required. In this case, assume that three workers are available to run a batch job for the parallel simulations. The job object returns useful metadata as shown. You can use the job ID to access the job object later from any machine. `NumWorkers` displays how many workers are running the simulations – the number of workers specified in the 'Pool' argument plus an additional head worker.

```matlab
simJob = batchsim(in,'Pool',3)
```

```
ID: 1
Type: pool
NumWorkers: 4
Username: #####
State: running
SubmitDateTime: ##-###-#### ##:##:##
StartDateTime: Running Duration: 0 days 0h 0m 0s
```

5. Now, wait for the job to finish before retrieving the outputs.
wait(simJob)

**Input Arguments**

**simJob** — Simulink.Simulation.Job object


Example: `simJob = batchsim(in,'Pool','5')`

**'stateOfJob'** — Value of the simulation job object State property to wait for

Value of the State property of Simulink.Simulation.Job object to wait for.

Example: `wait(simJob, 'queued')`

**timeout** — Maximum time to wait, in seconds

Specify a timeout for `wait` to block execution in seconds.

Example: `wait(simJob, 5)`

**See Also**

**Functions**

- `batchsim`
- `cancel`
- `diary`
- `fetchOutputs`
- `listAutoAttachedFiles`
- `parsim`

**Classes**

- `Simulink.Simulation.Job`
- `Simulink.SimulationInput`

**Topics**

- “Multiple Simulation Workflows”

**Introduced in R2018b**
Simulink.dialog.TreeControl class

Package: Simulink.dialog

Control Tree control programmatically

Description

Use an instance of Simulink.dialog.TreeControl class to perform the following operations:
- Get selected items from the tree
- Set selected items in the tree

Properties

Name
Uniquely identifies the dialog control and is a required field.

Type: character vector or string

Prompt
Specifies the text displayed on the tree dialog control.

Type: character vector or string

Default: empty

TreeItems
Specifies the items shown in the tree.

Type: cell array

Default: empty
**Callback**

Specifies the MATLAB command(s) to be executed when the dialog control is invoked.

**Type:** character vector or string  
**Default:** empty

**Row**

Specifies whether dialog control is placed on the current row or on a new row.

**Type:** character vector  
**Value:** 'current'|'new'  
**Default:** 'current'

**Enabled**

Indicates whether tree control is active on the mask dialog box.

**Type:** boolean  
**Value:** 'on'|'off'  
**Default:** 'on'

**Visible**

Indicates whether tree control is displayed on the mask dialog box.

**Type:** boolean  
**Value:** 'on'|'off'  
**Default:** 'on'

**Multiselect**

Indicates whether tree control is enabled for multiple selection.

**Type:** boolean
**Value**: 'on' | 'off'

**Default**: 'on'

**Tooltip**

Indicates whether tree control has a tooltip configured.

**Type**: character vector or string

**Default**: empty

**Methods**

- `setSelectedItems(TreeControl)`: Set values for a tree
- `getSelectedItems(TreeControl)`: Get the values of selected items from a tree hierarchy

**See Also**
setSelectedItems(TreeControl)

Class: Simulink.dialog.TreeControl
Package: Simulink.dialog

Set values for a tree

Syntax

treeControl.setSelectedItems({'Node1/ChildNode1', 'Node1/ChildNode2'})

Description

treeControl.setSelectedItems({'Node1/ChildNode1', 'Node1/ChildNode2'}) set the values for the selected items in the tree.

Input Arguments

TreeControl — tree handle
    table object

Handle to the tree, specified as an object. You can use the getDialogControl command to get the tree handle. For more information, see getDialogControl.

For example, treeControl = maskObj.getDialogControl('TreeControl'). Here, maskObj is the mask object and TreeControl is the name of the tree.

Data Types: cell array

value — value to be set
    cell-array of string

Value to be set for selected items in the tree.
Examples

maskObj = Simulink.Mask.get( gcb );
treeControl = maskObj.getDialogControl('Control2')

% Set selected items
treeControl.setSelectedItems({'Node1/ChildNode1', 'Node1/ChildNode2'})

See Also

“Create Block Masks” | Simulink.Mask

Introduced in R2019b
getSelectedItems(TreeControl)

Class: Simulink.dialog.TreeControl
Package: Simulink.dialog

Get the values of selected items from a tree hierarchy

Syntax

selectedItems = treeControl.getSelectedItems()

Description

selectedItems = treeControl.getSelectedItems() fetches the information of the selected items in the tree.

Output Arguments

selectedItems — value of selected items
cell array

Value of the selected items, returned as a cell array.

Examples

% Get block mask handle
maskObj = Simulink.Mask.get(gca);

% Get tree handle
treeControl = maskObj.getDialogControl('TreeControl');
treeControl.getSelectedItems();
ans =

4×1 cell array

{'Node1/ChildNode1'}
{'Node1/ChildNode2'}
{'Node2/ChildNode1'}
{'Node2/ChildNode2'}

See Also
“Create Block Masks” | Simulink.Mask

Introduced in R2019b
**getSimulationJobs**

Get all `Simulink.Simulation.Job` objects from cluster

**Syntax**

```
jobs = getSimulationJobs(myCluster)
```

**Description**

`jobs = getSimulationJobs(myCluster)` returns an array of `Simulink.Simulation.Job` objects that correspond to the jobs created by executing of `batchsim` on cluster, `myCluster`.

**Examples**

**Get a `Simulink.Simulation.Job` Object from the Cluster**

This example shows how to access a `batchsim` job that was submitted to a cluster. Assume that `myCluster` is a `parallel.Cluster` object on which the `Simulink.Simulation.Job` object is running.

```
jobs = getSimulationJobs(myCluster)
```

1×2 Job array:

<table>
<thead>
<tr>
<th>ID</th>
<th>Type</th>
<th>State</th>
<th>FinishDateTime</th>
<th>Username</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>queued</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>queued</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The output displays all the Simulink.Simulation.Job objects on cluster, myCluster.

**Input Arguments**

myCluster — parallel.Cluster object

Cluster object representing parallel cluster compute resources.

**Output Arguments**

jobs — Array of Simulink.Simulation.Job objects


**See Also**

**Functions**

batchsim | cancel | diary | fetchOutputs | listAutoAttachedFiles | parcluster | parsim | wait

**Classes**


**Topics**

“Multiple Simulation Workflows”
“Batch Processing” (Parallel Computing Toolbox)
“Job Monitor” (Parallel Computing Toolbox)

**Introduced in R2018b**
Model and Block Parameters

- “Model Parameters” on page 6-2
- “Common Block Properties” on page 6-112
- “Block-Specific Parameters” on page 6-132
- “Mask Parameters” on page 6-281
Model Parameters

In this section...

| “About Model Parameters” on page 6-2 |
| “Examples of Setting Model Parameters” on page 6-111 |

About Model Parameters

You can query and/or modify the properties (parameters) of a Simulink model from the command line. Parameters that describe a model are model parameters, and parameters that describe a Simulink block are block parameters. Block parameters that are common to Simulink blocks are called common block parameters. There are also block-specific parameters. Masks also have parameters, that is, parameters that describe a masked block.

The model and block properties can also include callbacks, which are commands that execute when certain model or block events occur. These events include opening a model, simulating a model, copying a block, opening a block, and so on.

This table lists, in alphabetical order, parameters that describe a model. You can set these parameters using the `set_param` command. The Description column indicates where you can set the value on a dialog box.

For examples, see “Examples of Setting Model Parameters” on page 6-111. The Values column shows the type of value required, the possible values (separated with a vertical line), and the default value enclosed in braces.

The table also includes model callback parameters (see “Callbacks for Customized Model Behavior”). Do not use model parameters in a `PreLoadFcn` callback. Instead, use them in a `PostLoadFcn` callback.
## Model Parameters in Alphabetical Order

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>AbsTol</td>
<td>Specify the largest acceptable solver error, as the value of the measured state approaches zero.</td>
<td>{'auto'}</td>
</tr>
<tr>
<td></td>
<td>Set by <strong>Absolute tolerance</strong> on the <strong>Solver</strong> pane of the Configuration Parameters dialog box.</td>
<td></td>
</tr>
<tr>
<td>AccelVerboseBuild</td>
<td>Controls the verbosity level during code generation for Simulink Accelerator mode, model reference Accelerator mode, and Rapid Accelerator mode.</td>
<td>{'off'}</td>
</tr>
<tr>
<td></td>
<td>Set by <strong>Verbose accelerator builds</strong> on the Configuration Parameters dialog box.</td>
<td></td>
</tr>
<tr>
<td>AlgebraicLoopMsg</td>
<td>Specifies diagnostic action to take when there is an algebraic loop.</td>
<td>'none'</td>
</tr>
<tr>
<td></td>
<td>Set by <strong>Algebraic loop</strong> on the <strong>Solver</strong> section of the <strong>Diagnostics</strong> pane of the Configuration Parameters dialog box.</td>
<td></td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
<td>Values</td>
</tr>
<tr>
<td>-----------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>-----------------------</td>
</tr>
<tr>
<td>ArrayBoundsChecking</td>
<td>Select the diagnostic action to take when blocks write data to locations outside the memory allocated to them.</td>
<td>{'none'}</td>
</tr>
<tr>
<td></td>
<td>Set by <strong>Array bounds exceeded</strong> on the Configuration Parameters dialog box.</td>
<td></td>
</tr>
<tr>
<td>ArtificialAlgebraicLoopMsg</td>
<td>Specifies diagnostic action to take if algebraic loop minimization cannot be performed for a subsystem because an input port of that subsystem has direct feedthrough.</td>
<td>'none'</td>
</tr>
<tr>
<td></td>
<td>Set by <strong>Minimize algebraic loop</strong> on the <strong>Solver</strong> section of the <strong>Diagnostics</strong> pane in the Configuration Parameters dialog box.</td>
<td></td>
</tr>
<tr>
<td>AssertControl</td>
<td>Enable model verification blocks in the current model either globally or locally.</td>
<td>{'UseLocalSettings'}</td>
</tr>
<tr>
<td></td>
<td>Set by <strong>Model Verification block enabling</strong> on the Configuration Parameters dialog box.</td>
<td></td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
<td>Values</td>
</tr>
<tr>
<td>--------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>-----------------------------------</td>
</tr>
<tr>
<td>AutoInsertRateTranBlk</td>
<td>Specify whether Simulink software inserts hidden Rate Transition blocks between blocks that have different sample rates. Set by Automatically handle rate transition for data transfer on the Solver pane of the Configuration Parameters dialog box.</td>
<td>'on'</td>
</tr>
<tr>
<td>BlockDescription-StringDataTip</td>
<td>Specifies whether to display the user description for a block as a data tip. In the Simulink Editor, in the Debug tab, select Information Overlays &gt; Description in Tooltip.</td>
<td>'on'</td>
</tr>
<tr>
<td>BlockNameDataTip</td>
<td>Specifies whether to display the block name as a data tip. In the Simulink Editor, in the Debug tab, select Information Overlays &gt; Name in Tooltip.</td>
<td>'on'</td>
</tr>
<tr>
<td>BlockParametersDataTip</td>
<td>Specifies whether to display a block parameter in a data tip. In the Simulink Editor, in the Debug tab, select Information Overlays &gt; Parameters in Tooltip.</td>
<td>'on'</td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
<td>Values</td>
</tr>
<tr>
<td>--------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>-----------------</td>
</tr>
<tr>
<td>BlockPriority-ViolationMsg</td>
<td>Select the diagnostic action to take if Simulink software detects a block priority specification error.</td>
<td>{'warning'}</td>
</tr>
<tr>
<td></td>
<td>Set by <strong>Block priority violation</strong> on the <strong>Solver</strong> section of the <strong>Diagnostics</strong> pane of the Configuration Parameters dialog box.</td>
<td></td>
</tr>
<tr>
<td>BlockReduction</td>
<td>Enables block reduction optimization.</td>
<td>{'on'}</td>
</tr>
<tr>
<td></td>
<td>Set by <strong>Block reduction</strong> on the Configuration Parameters dialog box.</td>
<td></td>
</tr>
<tr>
<td>BlockReductionOpt</td>
<td>See BlockReduction parameter for more information.</td>
<td></td>
</tr>
<tr>
<td>BooleanDataType</td>
<td>Enable Boolean mode.</td>
<td>{'on'}</td>
</tr>
<tr>
<td></td>
<td>Set by <strong>Implement logic signals as Boolean data (vs. double)</strong> on the Configuration Parameters dialog box.</td>
<td></td>
</tr>
<tr>
<td>BrowserLookUnderMasks</td>
<td>Show masked subsystems in the Model Browser.</td>
<td>'on'</td>
</tr>
<tr>
<td></td>
<td>On the <strong>Modeling</strong> tab, select <strong>Environment &gt; Model Browser</strong>. In the <strong>Model Browser</strong> pane, select <strong>≡ &gt; Systems with Mask Parameters.</strong></td>
<td></td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
<td>Values</td>
</tr>
<tr>
<td>----------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>-----------------------------</td>
</tr>
<tr>
<td>BrowserShowLibraryLinks</td>
<td>Show library links in the Model Browser.</td>
<td>'on'</td>
</tr>
<tr>
<td></td>
<td>On the Modeling tab, select Environment &gt; Model Browser. In the Model Browser pane, select ▼ → Library Links.</td>
<td></td>
</tr>
<tr>
<td>BufferReusableBoundary</td>
<td>Insert buffers at reusable subsystem boundaries if needed.</td>
<td>'on'</td>
</tr>
<tr>
<td>BufferReuse</td>
<td>Enable reuse of block I/O buffers.</td>
<td>{'on'}</td>
</tr>
<tr>
<td></td>
<td>Set by “Reuse local block outputs” (Simulink Coder) on the Optimization pane of the Configuration Parameters dialog box.</td>
<td></td>
</tr>
<tr>
<td>BusNameAdapt</td>
<td>Repair broken selections in the Bus Selector and Bus Assignment block parameters dialog boxes that are due to upstream bus hierarchy changes.</td>
<td>{'WarnAndRepair'}</td>
</tr>
<tr>
<td></td>
<td>Set by “Repair bus selections” on the Diagnostics &gt; Connectivity pane of the Configuration Parameters dialog box.</td>
<td></td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
<td>Values</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>-------------------------------</td>
</tr>
<tr>
<td><strong>BusObjectLabelMismatch</strong></td>
<td>Select the diagnostic action to take if the name of a bus element does not match the name specified by the corresponding bus object. Set by <strong>Element name mismatch</strong> on the <strong>Diagnostics &gt; Connectivity</strong> pane of the Configuration Parameters dialog box.</td>
<td>'none'</td>
</tr>
<tr>
<td><strong>CheckExecutionContext‐RuntimeOutputMsg</strong></td>
<td>Specify whether to display a warning if Simulink software detects potential output differences from previous releases. Set by <strong>Check runtime output of execution context</strong> on the Configuration Parameters dialog box.</td>
<td>'on'</td>
</tr>
<tr>
<td><strong>CheckForMatrix‐Singularity</strong></td>
<td>See <strong>CheckMatrixSingularityMsg</strong> parameter for more information.</td>
<td></td>
</tr>
<tr>
<td><strong>CheckMatrix‐SingularityMsg</strong></td>
<td>Select the diagnostic action to take if the Product block detects a singular matrix while inverting one of its inputs in matrix multiplication mode. Set by <strong>Division by singular matrix</strong> on the <strong>Data Validity Diagnostics</strong> pane of the Configuration Parameters dialog box.</td>
<td>{'none'}</td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
<td>Values</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td>-------------------------------------------------------------------------------------------------------</td>
<td>-------------------------------</td>
</tr>
<tr>
<td>CheckModelReference-TargetMessage</td>
<td>Select the diagnostic action to take if Simulink software detects a target that needs to be rebuilt.</td>
<td>'none'</td>
</tr>
<tr>
<td></td>
<td>Set by &quot;Never rebuild diagnostic&quot; on the Model Referencing pane of the Configuration Parameters dialog box.</td>
<td></td>
</tr>
<tr>
<td>CheckSSInitialOutputMsg</td>
<td>Enable checking for undefined initial subsystem output.</td>
<td>{'on'}</td>
</tr>
<tr>
<td></td>
<td>Set by Check undefined subsystem initial output on the Configuration Parameters dialog box.</td>
<td></td>
</tr>
<tr>
<td>CloseFcnc</td>
<td>Set the close callback function, which can be a command or a variable.</td>
<td>{''}</td>
</tr>
<tr>
<td></td>
<td>Set by Model close function on the Callbacks pane of the Model Properties dialog box.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>See &quot;Create Model Callbacks&quot; for more information.</td>
<td></td>
</tr>
<tr>
<td>CompiledModelBlockNormal ModeVisibility</td>
<td>For a top model that is being simulated or that is in a compiled state, return information about which Model blocks have normal mode visibility enabled.</td>
<td>Return values indicate which Model blocks have normal mode visibility enabled.</td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
<td>Values</td>
</tr>
<tr>
<td>----------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>---------------------</td>
</tr>
<tr>
<td>ConditionallyExecute-Inputs</td>
<td>Enable conditional input branch execution optimization.</td>
<td>{'on'}</td>
</tr>
<tr>
<td></td>
<td>Set by <strong>Conditional input branch execution</strong> on the Configuration Parameters dialog box.</td>
<td></td>
</tr>
<tr>
<td>ConsecutiveZCsStepRelTol</td>
<td>Relative tolerance associated with the time difference between zero-crossing events.</td>
<td>{'10<em>128</em>eps'}</td>
</tr>
<tr>
<td></td>
<td>Set by <strong>Time tolerance</strong> on the <strong>Solver</strong> pane of the Configuration Parameters dialog box.</td>
<td></td>
</tr>
<tr>
<td>ConsistencyChecking</td>
<td>Select the diagnostic action to take if S-functions have continuous sample times, but do not produce consistent results when executed multiple times.</td>
<td>{'none'}</td>
</tr>
<tr>
<td></td>
<td>Set by <strong>Solver data inconsistency</strong> on the <strong>Solver</strong> section of the <strong>Diagnostics</strong> pane of the Configuration Parameters dialog box.</td>
<td></td>
</tr>
<tr>
<td>ContinueFcn</td>
<td>Continue simulation callback.</td>
<td>{''}</td>
</tr>
<tr>
<td></td>
<td>Set by <strong>Simulation continue function</strong> on the <strong>Callbacks</strong> pane of the Model Properties dialog box.</td>
<td></td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
<td>Values</td>
</tr>
<tr>
<td>--------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>--------</td>
</tr>
<tr>
<td>CovCompData</td>
<td>If CovHtmlReporting is set to on and CovCumulativeReport is set to on, this parameter specifies cvdata objects containing additional model coverage data to include in the model coverage report. Set by the Additional data to include in report field in the Advanced parameters section of the Coverage &gt; Results pane of the Configuration Parameters dialog box.</td>
<td>'{'}</td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
<td>Values</td>
</tr>
<tr>
<td>----------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>-------------------------</td>
</tr>
<tr>
<td>CovCumulativeReport</td>
<td>If CovHtmlReporting is set to on, this parameter allows the CovCumulativeReport and CovCompData parameters to specify the number of coverage results displayed in the model coverage report. If set to on, the Simulink Coverage™ software displays the coverage results from successive simulations in the report. If set to off, the software displays the coverage results for the last simulation in the report. Set by the <strong>Include cumulative data in coverage report</strong> option in the <strong>Advanced parameters</strong> section of the <strong>Coverage &gt; Results</strong> pane of the Configuration Parameters dialog box.</td>
<td>'on'</td>
</tr>
<tr>
<td>CovDataFileName</td>
<td>If CovEnable is set to on, specifies the name of the file to which Simulink Coverage saves the coverage data results.</td>
<td>{''}</td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
<td>Values</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>-------------------------</td>
</tr>
<tr>
<td>CovCumulativeVarName</td>
<td>If CovSaveCumulativeWorkspaceVar is set to on, the Simulink Coverage software saves the results of successive simulations in the workspace variable specified by this property.</td>
<td>{'covCumulativeData'}</td>
</tr>
<tr>
<td>CovEnable</td>
<td>Enables coverage analysis for Simulink Coverage. Set by <strong>Enable coverage analysis</strong> on the <strong>Coverage</strong> pane of the Configuration Parameters dialog box.</td>
<td>'on'</td>
</tr>
<tr>
<td>CovEnableCumulative</td>
<td>Accumulates model coverage results for Simulink Coverage from successive simulations. Set this and CovSaveCumulativeWorkspaceVar to on to collect model coverage results for multiple simulations in one cvdata object.</td>
<td>'on'</td>
</tr>
<tr>
<td>CovExternalEMLEnable</td>
<td>Enables coverage for any external MATLAB functions that MATLAB functions for code generation call in your model. The functions can be defined in a MATLAB Function block or in a Stateflow chart. Enable this feature by checking <strong>MATLAB Files</strong> on the <strong>Coverage</strong> pane of the Configuration Parameters dialog box.</td>
<td>{'on'}</td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
<td>Values</td>
</tr>
<tr>
<td>-----------</td>
<td>-------------</td>
<td>--------</td>
</tr>
<tr>
<td>CovForceBlockReductionOff</td>
<td>If CovForceBlockReductionOff is set to on, the Simulink Coverage software ignores the value of the Simulink <strong>Block reduction</strong> parameter. The software provides coverage data for every block in the model that collects coverage.</td>
<td>{'on'}</td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
<td>Values</td>
</tr>
<tr>
<td>-----------------</td>
<td>------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>-------------------</td>
</tr>
<tr>
<td>CovHighlightResults</td>
<td>Enable model coloring for coverage results. Enabled by selecting <strong>Display coverage results using model coloring</strong> on the <strong>Coverage &gt; Results</strong> pane of the Configuration Parameters dialog box.</td>
<td>'on'</td>
</tr>
</tbody>
</table>

**Note** If you use the toolbar buttons to simulate a model with coverage enabled, this setting is not honored and the model coloring for coverage results always appears after each simulation. You can click **Highlight model with coverage results** in the Results Explorer to enable or disable model coverage highlighting. You access the Results Explorer by selecting **Analysis > Coverage > Open Results Explorer**. For more information, see “Accessing Coverage Data from the Results Explorer” (Simulink Coverage).

If you enabled the Simulink toolstrip tech preview, you enable an disable model highlighting from the Simulink Coverage contextual tabs, which appear when you open the **Coverage Analyzer** app,
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>under Verification, Validation, and Test.</td>
<td></td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
<td>Values</td>
</tr>
<tr>
<td>------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| CovHTMLOptions   | If CovHtmlReporting is set to on, use this parameter to select from a set of display options for the resulting model coverage report. Select these options in the Results Explorer settings after you record coverage for a model. | Character vector of appended character sets separated by a space. HTML options are enabled or disabled through a value of 1 or 0, respectively, in the following character sets (default values shown):  
• '-sRT=1' — Show report  
• '-sVT=0' — Web view mode  
• '-aTS=1' — Include each test in the model summary  
• '-bRG=1' — Produce bar graphs in the model summary  
• '-bTC=0' — Use two color bar graphs (red, blue)  
• '-hTR=0' — Display hit/count ratio in the model summary  
• '-xEM=0' — Exclude execution metric details from report  
• '-nFC=0' — Exclude fully covered model objects from report  
• '-nFD=1' — Exclude fully covered model object details from report  
• '-scm=1' — Include cyclomatic complexity numbers in summary  
• '-bcm=1' — Include cyclomatic complexity numbers in block details  
• '-xEv=0' — Filter Stateflow events from report |
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>CovIncludeTopModel</td>
<td>Option to include the top-level model in the coverage analysis.</td>
<td>{'on'}</td>
</tr>
<tr>
<td>CovHtmlReporting</td>
<td>Set to on to tell the Simulink Coverage software to create an HTML report containing the coverage data at the end of simulation.</td>
<td>'on'</td>
</tr>
</tbody>
</table>

Set by **Generate report automatically after analysis** on the **Coverage > Results** pane of the Configuration Parameters dialog box.

**Note** If you enable the Simulink Toolstrip tech preview and you use the toolbar buttons to simulate a model with coverage enabled, the HTML report does not appear after a simulation. You access the HTML report from the Simulink Coverage contextual tabs, which appear when you open the **Coverage Analyzer** app, under **Verification, Validation, and Test**.
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>CovLogicBlockShortCircuit</td>
<td>Enables the option to treat Simulink logic blocks as short-circuited for coverage analysis. Enabled by selecting <strong>Treat Simulink logic blocks as short-circuited</strong> on the <strong>Coverage</strong> pane of the Configuration Parameters dialog box.</td>
<td>'on'</td>
</tr>
<tr>
<td>CovMcdcMode</td>
<td>Determines the definition of Modified Condition Decision Coverage (MCDC) to use during coverage analysis. To record model coverage using the masking MCDC definition, set <strong>CovMcdcMode</strong> to 'Masking'. To record model coverage using the unique-cause MCDC definition, set <strong>CovMcdcMode</strong> to 'UniqueCause'.</td>
<td>{'Masking'}</td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
<td>Values</td>
</tr>
<tr>
<td>----------------------</td>
<td>----------------------------------------------------------------------------------------------</td>
<td>-----------------------</td>
</tr>
<tr>
<td>CovMetricSettings</td>
<td>Selects coverage metrics for a coverage report. Coverage metrics are enabled by selecting the check boxes for individual coverages in the <strong>Coverage metrics</strong> section of the <strong>Coverage</strong> pane of the Configuration Parameters dialog box. Enable options 's' and 'w' by selecting <strong>Treat Simulink Logic blocks as short-circuited</strong> and <strong>Warn when unsupported blocks exist in model</strong>, respectively, on the <strong>Coverage</strong> pane of the Configuration Parameters dialog box. Disable option 'e' by selecting <strong>Display coverage results using model coloring</strong> in the <strong>Results Explorer</strong> settings after you record coverage for a model.</td>
<td>{'dwe'}</td>
</tr>
</tbody>
</table>

Each order-independent value enables a coverage metric or option as follows:
- 'd' — Enable decision coverage
- 'c' — Enable condition coverage and decision coverage
- 'm' — Enable MCDC coverage, condition coverage, and decision coverage
- 't' — Enable lookup table coverage
- 'r' — Enable signal range coverage
- 'z' — Enable signal size coverage
- 'o' — Enable coverage for Simulink Design Verifier blocks
- 'i' — Enable saturation on integer overflow coverage
- 'b' — Enable relational boundary coverage
- 's' — Treat Simulink logic blocks as short-circuited
- 'w' — Warn when unsupported blocks exist in model
- 'e' — Eliminate model coloring for coverage results

**Note** The metrics and options set by this parameter can also be set by the following parameters:
- CovHighlightResults
- CovLogicBlockShortCircuit
- CovMetricStructuralLevel
- CovMetricLookupTable

6-20
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>CovMetricSignalRange</td>
<td>Enable lookup table coverage. Enabled by selecting <strong>Lookup Table</strong> in the <strong>Coverage metrics</strong> section of the <strong>Coverage</strong> pane of the Configuration Parameters dialog box.</td>
<td>'on'</td>
</tr>
<tr>
<td>CovMetricSignalSize</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CovMetricObjectiveConstraint</td>
<td>Enable Simulink Design Verifier objectives and constraints coverage. Enabled by selecting <strong>Objectives and Constraints</strong> in the <strong>Coverage metrics</strong> section of the <strong>Coverage</strong> pane of the Configuration Parameters dialog box.</td>
<td>'on'</td>
</tr>
<tr>
<td>CovMetricSaturateOnIntegerOverflow</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CovMetricRelationalBoundary</td>
<td>Enable relational boundary coverage. Enabled by selecting <strong>Relational Boundary</strong> in the <strong>Coverage metrics</strong> section of the <strong>Coverage</strong> pane of the Configuration Parameters dialog box.</td>
<td>'on'</td>
</tr>
<tr>
<td>CovUnsupportedBlockWarning</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
<td>Values</td>
</tr>
<tr>
<td>-----------------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>-----------------------------</td>
</tr>
<tr>
<td>CovMetricSaturateOnIntegerOverflow</td>
<td>Enable saturate on integer overflow coverage. Enabled by selecting <strong>Saturate on Integer Overflow</strong> in the <strong>Coverage metrics</strong> section of the <strong>Coverage</strong> pane of the Configuration Parameters dialog box.</td>
<td>'on'</td>
</tr>
<tr>
<td>CovMetricSignalRange</td>
<td>Enable signal range coverage. Enabled by selecting <strong>Signal Range</strong> in the <strong>Coverage metrics</strong> section of the <strong>Coverage</strong> pane of the Configuration Parameters dialog box.</td>
<td>'on'</td>
</tr>
<tr>
<td>CovMetricSignalSize</td>
<td>Enable signal size coverage. Enabled by selecting <strong>Signal Size</strong> in the <strong>Coverage metrics</strong> section of the <strong>Coverage</strong> pane of the Configuration Parameters dialog box.</td>
<td>'on'</td>
</tr>
<tr>
<td>CovMetricStructuralLevel</td>
<td>Define the level of structural coverage. Set by <strong>Structural coverage level</strong> on the <strong>Coverage</strong> pane of the Configuration Parameters dialog box.</td>
<td>'BlockExecution'</td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
<td>Values</td>
</tr>
<tr>
<td>----------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>-----------------</td>
</tr>
<tr>
<td>CovModelRefEnable</td>
<td>If <code>CovModelRefEnable</code> is set to <code>on</code> or <code>all</code>, the Simulink Coverage software generates coverage data for all referenced models. If <code>CovModelRefEnable</code> is set to <code>filtered</code>, coverage data is collected for all referenced models except those specified by the parameter <code>CovModelRefExcluded</code>.</td>
<td>'on'</td>
</tr>
<tr>
<td></td>
<td>IFIC <code>CovModelRefEnable</code> is set to <code>filtered</code>, this parameter stores a comma-separated list of referenced models for which coverage is disabled.</td>
<td>{''}</td>
</tr>
<tr>
<td>CovModelRefExcluded</td>
<td>If <code>CovModelRefEnable</code> is set to <code>filtered</code>, this parameter stores a comma-separated list of referenced models for which coverage is disabled.</td>
<td>{''}</td>
</tr>
<tr>
<td></td>
<td>IFIC <code>CovModelRefEnable</code> is set to <code>filtered</code>, this parameter stores a comma-separated list of referenced models for which coverage is disabled.</td>
<td>{''}</td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
<td>Values</td>
</tr>
<tr>
<td>---------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>---------------------</td>
</tr>
<tr>
<td>CovNameIncrementing</td>
<td>If CovSaveSingleToWorkspaceVar is set to on, setting CovNameIncrementing to on causes the Simulink Coverage software to append numerals to the workspace variable names for results so that earlier results are not overwritten (for example, covdata1, covdata2, etc.)</td>
<td>'on'</td>
</tr>
<tr>
<td></td>
<td>Set by <strong>Increment variable name with each simulation</strong> below the selected <strong>Save last run in workspace variable</strong> check box on the <strong>Coverage &gt; Results</strong> pane of the Configuration Parameters dialog box.</td>
<td></td>
</tr>
<tr>
<td>CovOutputDir</td>
<td>If CovEnable is set to on, specifies the directory in which Simulink Coverage saves the coverage output files.</td>
<td>{''}</td>
</tr>
<tr>
<td>CovPath</td>
<td>Model path of the subsystem for which the Simulink Coverage software gathers and reports coverage data.</td>
<td>{''}</td>
</tr>
<tr>
<td></td>
<td>Set by selecting <strong>Subsystem</strong> on the <strong>Coverage</strong> pane of the Configuration Parameters dialog box and then clicking <strong>Select Subsystem</strong>.</td>
<td></td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
<td>Values</td>
</tr>
<tr>
<td>------------------------------</td>
<td>-----------------------------------------------------------------------------------------------</td>
<td>----------------------</td>
</tr>
<tr>
<td>CovReportOnPause</td>
<td>Specifies that when you pause during simulation, the model coverage report appears in updated form, with coverage results up to the current pause or stop time.</td>
<td>{'on'}</td>
</tr>
<tr>
<td>CovSaveCumulativeToWorkspaceVar</td>
<td>If set to on, the Simulink Coverage software accumulates and saves the results of successive simulations in the workspace variable specified by CovCumulativeVarName.</td>
<td>'on'</td>
</tr>
<tr>
<td>CovSaveName</td>
<td>If CovSaveSingleToWorkspaceVar is set to on, the Simulink Coverage software saves the results of the last simulation run in the workspace variable specified by this property.</td>
<td>{'covdata'}</td>
</tr>
</tbody>
</table>

Set by **cvdata object name** below the selected **Save last run in workspace variable** check box on the **Coverage > Results** pane of the Configuration Parameters dialog box.
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>CovSaveSingleTo-WorkspaceVar</td>
<td>If set to on, the Simulink Coverage software saves the results of the last simulation run in the workspace variable specified by CovSaveName. Set by <strong>Save last run in workspace variable</strong> on the <strong>Coverage &gt; Results</strong> pane of the Configuration Parameters dialog box.</td>
<td>'on'</td>
</tr>
<tr>
<td>CovScope</td>
<td>Sets the scope of analysis for coverage recording. Set by the <strong>Scope of analysis section</strong> of the <strong>Coverage</strong> pane in the Configuration Parameters dialog box.</td>
<td>{'EntireSystem'}</td>
</tr>
<tr>
<td>CovSFcnEnable</td>
<td>Enables coverage for C/C++ S-Function blocks in your model. Enable this feature by checking <strong>C/C++ S-Functions</strong> on the <strong>Coverage</strong> pane of the Configuration Parameters dialog box. For more information, see “Coverage for Custom C/C++ Code in Simulink Models” (Simulink Coverage).</td>
<td>'on'</td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
<td>Values</td>
</tr>
<tr>
<td>--------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>-----------------------</td>
</tr>
<tr>
<td>CovShowResultsExplorer</td>
<td>Option to shows the results explorer after simulation. Enabled by selecting <strong>Show Results Explorer</strong> on the <strong>Coverage &gt; Results</strong> pane of the Configuration Parameters dialog box.</td>
<td>{'on'}</td>
</tr>
<tr>
<td></td>
<td><strong>Note</strong> If you enable the Simulink Toolstrip tech preview and you use the toolbar buttons to simulate a model with coverage enabled, the Results Explorer does not appear after a simulation. You can access the Results Explorer from the Simulink Coverage contextual tabs, which appear when you open the <strong>Coverage Analyzer</strong> app, under <strong>Verification, Validation, and Test</strong>.</td>
<td></td>
</tr>
<tr>
<td>CovUnsupportedBlockWarning</td>
<td>Warn when unsupported blocks exist in model. Enabled by selecting <strong>Warn when unsupported blocks exist in model</strong> on the <strong>Coverage</strong> pane of the Configuration Parameters dialog box.</td>
<td>'on'</td>
</tr>
</tbody>
</table>

**Model Parameters**

6-27
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Created</td>
<td>Date and time model was created.</td>
<td>character vector</td>
</tr>
<tr>
<td></td>
<td>Set by <strong>Created on</strong> on the <strong>History</strong> pane of the Model Properties dialog box.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>See “Model Information” for more information.</td>
<td></td>
</tr>
<tr>
<td>Creator</td>
<td>Name of model creator.</td>
<td>character vector</td>
</tr>
<tr>
<td></td>
<td>Set by <strong>Created by</strong> on the <strong>History</strong> pane of the Model Properties dialog box.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>See “Model Information” for more information.</td>
<td></td>
</tr>
<tr>
<td>CurrentBlock</td>
<td>For internal use.</td>
<td></td>
</tr>
<tr>
<td>CurrentOutputPort</td>
<td>For internal use.</td>
<td></td>
</tr>
<tr>
<td>CurrentOperatingPoint</td>
<td>Save the operating point of the simulation if the simulation is in a paused state.</td>
<td>Simulink.op.ModelOperatingPoint object</td>
</tr>
<tr>
<td>DataDictionary</td>
<td>Simulink data dictionary to which this model is linked.</td>
<td>{''}</td>
</tr>
<tr>
<td></td>
<td>Set by <strong>Data Dictionary</strong> and <strong>Base Workspace</strong> on the <strong>Data</strong> pane of the Model Properties dialog box.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>For basic information about data dictionaries, see “What Is a Data Dictionary?”. To use this parameter programmatically, see “Store Data in Dictionary Programmatically”.</td>
<td></td>
</tr>
<tr>
<td><strong>Parameter</strong></td>
<td><strong>Description</strong></td>
<td><strong>Values</strong></td>
</tr>
<tr>
<td>---------------------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>---------------------------------------------------------------------------</td>
</tr>
<tr>
<td>DataLoggingOverride</td>
<td>A Simulink.SimulationData.ModelLoggingInfo object that specifies the signal logging override settings for a model.</td>
<td>Simulink.SimulationData.ModelLoggingInfo — {'OverrideSignals'}</td>
</tr>
<tr>
<td></td>
<td>See “Override Signal Logging Settings”.</td>
<td></td>
</tr>
<tr>
<td>DatasetSignalFormat</td>
<td>Format for logged Dataset leaf elements.</td>
<td>'timetable'</td>
</tr>
<tr>
<td></td>
<td>For details, see “Dataset signal format”.</td>
<td></td>
</tr>
<tr>
<td>DataTransfer</td>
<td>A Simulink.GlobalDataTransfer object that configures data transfers for models configured for concurrent execution.</td>
<td>'on'</td>
</tr>
<tr>
<td>DataTypeOverride</td>
<td>Specifies data type used to override fixed-point data types.</td>
<td>{'UseLocalSettings'}</td>
</tr>
<tr>
<td></td>
<td>Set by <strong>Data type override</strong> on the Fixed-Point Tool.</td>
<td></td>
</tr>
<tr>
<td>Decimation</td>
<td>Specify that Simulink software output only every N points, where N is the specified decimation factor.</td>
<td>'1'</td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
<td>Values</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>-----------------------------</td>
</tr>
<tr>
<td>DefaultParameterBehavior</td>
<td>Enable inlining of block parameters in generated code. Set by Default parameter behavior, see “Default parameter behavior” (Simulink Coder).</td>
<td>'Inlined'</td>
</tr>
<tr>
<td>DefaultAnnotationFontName</td>
<td>Name of font to use for new annotation text, block text, or signal line labels and on existing annotations, block names, or signal lines whose FontName property is set to 'auto'. To set, in the Format tab, click Fonts for Model.</td>
<td>character vector</td>
</tr>
<tr>
<td>DefaultBlockFontName</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DefaultLineFontName</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DefaultAnnotationFontSize</td>
<td>Size of font to use for new annotation text, block text, or signal line labels and on existing annotations, blocks, or signal lines whose FontSize property is set to -1. To set, in the Format tab, click Fonts for Model.</td>
<td>positive integer</td>
</tr>
<tr>
<td>DefaultBlockFontSize</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DefaultLineFontSize</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DefaultAnnotationFontAngle</td>
<td>Angle of font for new annotation text, block text, or signal line labels and on existing annotations, blocks, or signal lines whose FontAngle property is set to 'auto'. To set, in the Format tab, click Fonts for Model.</td>
<td>{'normal'}</td>
</tr>
<tr>
<td>DefaultBlockFontAngle</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DefaultLineFontAngle</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
<td>Values</td>
</tr>
<tr>
<td>-----------------------------------</td>
<td>-----------------------------------------------------------------------------------------------</td>
<td>-----------------------------</td>
</tr>
<tr>
<td>DefaultAnnotationFontWeight</td>
<td>Weight of font for new annotation text, block text, or signal line labels and on existing annotations, blocks, or signal lines whose FontWeight property is set to 'auto'. To set, in the Format tab, click Fonts for Model.</td>
<td>{'normal'}</td>
</tr>
<tr>
<td>DefaultBlockFontWeight</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DefaultLineFontWeight</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DefaultUnderspecifiedDataType</td>
<td>Specify data type to use if Simulink cannot infer the type of a signal during data type propagation. Set by “Default for underspecified data type” on the Math and Data Types pane of the Configuration Parameters dialog box.</td>
<td>{'double'}</td>
</tr>
<tr>
<td>DeleteChildFcn</td>
<td>Delete child callback function. Created on the Callbacks pane of the Block Properties dialog box. See “Specify Block Callbacks” for more information.</td>
<td>'{}'</td>
</tr>
<tr>
<td>Description</td>
<td>Description of this model. Set by Model description on the Description pane of the Model Properties dialog box.</td>
<td>'{}'</td>
</tr>
<tr>
<td>Dirty</td>
<td>If the parameter is on, the model has unsaved changes.</td>
<td>'on'</td>
</tr>
<tr>
<td>DiscreteInheritContinuousMsg</td>
<td>For internal use.</td>
<td></td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
<td>Values</td>
</tr>
<tr>
<td>---------------------------------</td>
<td>--------------------------</td>
<td>---------------------------------------------</td>
</tr>
<tr>
<td>DisplayBdSearchResults</td>
<td>For internal use.</td>
<td></td>
</tr>
<tr>
<td>DisplayBlockIO</td>
<td>For internal use.</td>
<td></td>
</tr>
<tr>
<td>DisplayCallgraph-Dominators</td>
<td>For internal use</td>
<td></td>
</tr>
<tr>
<td>DisplayCompileStats</td>
<td>For internal use</td>
<td></td>
</tr>
<tr>
<td>DisplayCondInputTree</td>
<td>For internal use</td>
<td></td>
</tr>
<tr>
<td>DisplayCondStIdTree</td>
<td>For internal use</td>
<td></td>
</tr>
<tr>
<td>DisplayErrorDirections</td>
<td>For internal use</td>
<td></td>
</tr>
<tr>
<td>DisplayInvisibleSources</td>
<td>For internal use</td>
<td></td>
</tr>
<tr>
<td>DisplaySortedLists</td>
<td>For internal use</td>
<td></td>
</tr>
<tr>
<td>DisplayVectorAndFunctionCounts</td>
<td>For internal use</td>
<td></td>
</tr>
<tr>
<td>DisplayVectorPropagationResults</td>
<td>For internal use</td>
<td></td>
</tr>
<tr>
<td>EnableAccessToBaseWorkspace</td>
<td>Enables model access to base workspace. Must be set to 'on' if the model is not linked to a data dictionary.</td>
<td>{'on'}</td>
</tr>
<tr>
<td>ExecutionContextIcon</td>
<td>Show execution context bars on conditional subsystems that do not propagate execution context across the subsystem boundaries. In the Simulink Editor, in the <strong>Debug</strong> tab, select <strong>Information Overlays</strong> &gt; <strong>Execution Context</strong>.</td>
<td>'on'</td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
<td>Values</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>------------------------------------------------------------------------------</td>
<td>---------------------------------------------</td>
</tr>
<tr>
<td>ExplicitPartitioning</td>
<td>Specifies whether or not to manually map tasks (explicit mapping) or use the rate-based tasks.</td>
<td>'on'</td>
</tr>
<tr>
<td>ExpressionFolding</td>
<td>Enables expression folding.</td>
<td>{'on'}</td>
</tr>
<tr>
<td></td>
<td>Set by “<strong>Eliminate superfluous local variables (Expression folding)</strong>” (Simulink Coder) on the Configuration Parameters dialog box.</td>
<td></td>
</tr>
<tr>
<td>ExternalInput</td>
<td>Names of MATLAB workspace variables used to designate data and times to be loaded from the workspace.</td>
<td>{'[t, u]'}</td>
</tr>
<tr>
<td></td>
<td>Set by the <strong>Input</strong> field on the <strong>Data Import/Export</strong> pane of the Configuration Parameters dialog box.</td>
<td></td>
</tr>
<tr>
<td>ExtMode...</td>
<td>Parameters whose names start with <strong>ExtMode</strong> apply to Simulink external mode simulations.</td>
<td>For more information, see “Host-Target Communication with External Mode Simulation” (Simulink Coder).</td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
<td>Values</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>----------</td>
</tr>
<tr>
<td>ExtrapolationOrder</td>
<td>Extrapolation order of the <code>ode14x</code> implicit fixed-step solver.</td>
<td>integer — 1</td>
</tr>
<tr>
<td></td>
<td>Set by <strong>Extrapolation order</strong> on the <strong>Solver</strong> pane of the Configuration Parameters dialog box.</td>
<td></td>
</tr>
<tr>
<td>FastRestart</td>
<td>Enable or disable fast restart mode.</td>
<td>{'on'}</td>
</tr>
<tr>
<td></td>
<td>In the Simulink Editor toolbar, click the <strong>Fast restart</strong> button on or off.</td>
<td></td>
</tr>
<tr>
<td>FcnCallInpInside-ContextMsg</td>
<td>Specifies diagnostic action to take when Simulink software must compute any function-call subsystem inputs directly or indirectly during execution of a call to a function-call subsystem.</td>
<td>{'Error'}</td>
</tr>
<tr>
<td></td>
<td>Set by <strong>Context-dependent inputs</strong> on the <strong>Connectivity Diagnostics</strong> pane of the Configuration Parameters dialog box.</td>
<td></td>
</tr>
<tr>
<td>FileName</td>
<td>For internal use.</td>
<td></td>
</tr>
<tr>
<td>FinalStateName</td>
<td>Names of final states to save to the workspace after a simulation ends.</td>
<td>{'xFinal'}</td>
</tr>
<tr>
<td></td>
<td>Set by the <strong>Final states</strong> field on the <strong>Data Import/Export</strong> pane of the Configuration Parameters dialog box.</td>
<td></td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
<td>Values</td>
</tr>
<tr>
<td>-----------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>---------------------------------------</td>
</tr>
<tr>
<td>FixedStep</td>
<td>Fixed-step size.</td>
<td>{'auto'}</td>
</tr>
<tr>
<td></td>
<td>Set by Fixed-step size (fundamental sample time) on the Solver pane of the Configuration Parameters dialog box.</td>
<td></td>
</tr>
<tr>
<td>FixptConstOverflowMsg</td>
<td>Specifies diagnostic action to take when a fixed-point constant overflow occurs during simulation.</td>
<td>{'none'}</td>
</tr>
<tr>
<td></td>
<td>Set by Detect overflow on the Type Conversion Diagnostics pane of the Configuration Parameters dialog box.</td>
<td></td>
</tr>
<tr>
<td>FixptConstPrecisionLossMsg</td>
<td>Specifies diagnostic action to take when a fixed-point constant precision loss occurs during simulation.</td>
<td>{'none'}</td>
</tr>
<tr>
<td></td>
<td>Set by Detect precision loss on the Type Conversion Diagnostics pane of the Configuration Parameters dialog box.</td>
<td></td>
</tr>
<tr>
<td>FixptConstUnderflowMsg</td>
<td>Specifies diagnostic action to take when a fixed-point constant underflow occurs during simulation.</td>
<td>{'none'}</td>
</tr>
<tr>
<td></td>
<td>Set by Detect underflow on the Type Conversion Diagnostics pane of the Configuration Parameters dialog box.</td>
<td></td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
<td>Values</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td>------------------------------------------------------------------------------</td>
<td>-----------------------------</td>
</tr>
<tr>
<td>FixPtInfo</td>
<td>For internal use.</td>
<td></td>
</tr>
<tr>
<td>FollowLinksWhenOpeningFromGotoBlocks</td>
<td>Specifies whether to search for Goto tags in libraries referenced by the model when opening the From block dialog box.</td>
<td>'on'</td>
</tr>
<tr>
<td>ForceArrayBoundsChecking</td>
<td>For internal use.</td>
<td></td>
</tr>
<tr>
<td>ForceConsistencyChecking</td>
<td>For internal use.</td>
<td></td>
</tr>
<tr>
<td>ForceModelCoverage</td>
<td>For internal use.</td>
<td></td>
</tr>
<tr>
<td>ForwardingTable</td>
<td>Specifies the forwarding table for this library.</td>
<td>{{'old_path_1', 'new_path_1'} ... {'old_path_n', 'new_path_n'}}</td>
</tr>
<tr>
<td>ForwardingTableString</td>
<td>For internal use.</td>
<td></td>
</tr>
<tr>
<td>GeneratePreprocessorConditionals</td>
<td>When generating code for an ERT target, this parameter determines whether variant choices are enclosed within C preprocessor conditional statements (#if). When you select this option, Simulink analyzes all variant choices during an update diagram or simulation. This analysis provides early validation of the code generation readiness of all variant choices.</td>
<td>{'off'}</td>
</tr>
<tr>
<td>GridSpacing</td>
<td>Has no effect in Simulink Editor. This parameter will be removed in a future release.</td>
<td>integer — {20}</td>
</tr>
<tr>
<td>Handle</td>
<td>Handle of the block diagram for this model.</td>
<td>double</td>
</tr>
</tbody>
</table>

6-36
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>HardwareBoard</td>
<td>Select the type of hardware on which to run your model.</td>
<td>{'none'}</td>
</tr>
<tr>
<td></td>
<td>Set by &quot;<strong>Hardware board</strong>&quot; on the <strong>Hardware Implementation</strong> pane of the Configuration Parameters dialog box.</td>
<td></td>
</tr>
<tr>
<td>HideAutomaticNames</td>
<td>Hides block names given automatically by the Simulink Editor. See “Hide or Display Block Names”.</td>
<td>{'on'}</td>
</tr>
<tr>
<td>HiliteAncestors</td>
<td>For internal use.</td>
<td></td>
</tr>
<tr>
<td>IgnoreBidirectionalLines</td>
<td>For internal use.</td>
<td></td>
</tr>
<tr>
<td>IgnoredZcDiagnostic</td>
<td>Control diagnostic messages related to zero-crossings that are being ignored.</td>
<td>'none'</td>
</tr>
<tr>
<td>InheritedTsInSrcMsg</td>
<td>Message behavior when the sample time is inherited.</td>
<td>'none'</td>
</tr>
<tr>
<td></td>
<td>Set by <strong>Source block specifies -1 sample time</strong> on the <strong>Sample Time Diagnostics</strong> pane of the Configuration Parameters dialog box.</td>
<td></td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
<td>Values</td>
</tr>
<tr>
<td>-------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>-------------------</td>
</tr>
<tr>
<td>InitFcn</td>
<td>Function that is called when this model is first compiled for simulation.</td>
<td>{''}</td>
</tr>
<tr>
<td></td>
<td>Set by Model initialization function on the Callbacks pane of the Model Properties dialog box.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>See “Create Model Callbacks” for more information.</td>
<td></td>
</tr>
<tr>
<td>InitialState</td>
<td>Initial state name or values.</td>
<td>variable or vector — {'xInitial'}</td>
</tr>
<tr>
<td></td>
<td>Set by the Initial state field on the Data Import/Export pane of the Configuration Parameters dialog box.</td>
<td></td>
</tr>
<tr>
<td>InitialStep</td>
<td>Initial step size.</td>
<td>{'auto'}</td>
</tr>
<tr>
<td></td>
<td>Set by Initial step size on the Solver pane of the Configuration Parameters dialog box.</td>
<td></td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
<td>Values</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>---------------------------------------------------</td>
</tr>
<tr>
<td>InitInArrayFormatMsg</td>
<td>Message behavior when the initial state is an array. You set with the initial state with the <strong>Initial state</strong> configuration parameter. Avoid using an array for the initial state. If the order of the elements in the array does not match the order in which blocks initialize, the simulation can produce unexpected results. To promote deterministic simulation results, use the default setting or set the diagnostic to <strong>error</strong>. Alternatively, you can set the message behavior using the &quot;<strong>Initial state is array</strong>&quot; on the Configuration Parameters dialog box.</td>
<td>'none'</td>
</tr>
<tr>
<td>InsertRTBMode</td>
<td>Control whether the Rate Transition block parameter <strong>Ensure deterministic data transfer (maximum delay)</strong> is set for auto-inserted Rate Transition blocks. Set by <strong>Deterministic data transfer</strong> on the <strong>Solver</strong> pane of the Configuration Parameters dialog box.</td>
<td>'Always'</td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
<td>Values</td>
</tr>
<tr>
<td>------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>---------------------------------</td>
</tr>
<tr>
<td>InspectSignalLogs</td>
<td>Enable Simulink software to display logged signals in the Simulation Data Inspector tool at the end of a simulation or whenever you pause the simulation. Set by “Record logged workspace data in Simulation Data Inspector” on the Data Import/Export pane of the Configuration Parameters dialog box.</td>
<td>'on'</td>
</tr>
<tr>
<td>InstrumentedSignals</td>
<td>Returns a Simulink.HMI.InstrumentedSignals object with properties indicating the model name and the number of signals that are marked for logging. You can use the InstrumentedSignals object as a specification to configure a set of signals for logging.</td>
<td>Simulink.HMI.InstrumentedSignals</td>
</tr>
<tr>
<td>Int32ToFloatConvMsg</td>
<td>Specify message behavior when a 32-bit integer is converted to a single-precision float. Set by 32-bit integer to single precision float conversion on the Type Conversion Diagnostics pane of the Configuration Parameters dialog box.</td>
<td>'none'</td>
</tr>
<tr>
<td><strong>Parameter</strong></td>
<td><strong>Description</strong></td>
<td><strong>Values</strong></td>
</tr>
<tr>
<td>---------------</td>
<td>----------------</td>
<td>-----------</td>
</tr>
<tr>
<td><strong>IntegerOverflowMsg</strong></td>
<td>Specify message behavior when an integer overflow occurs.</td>
<td>'none'</td>
</tr>
<tr>
<td></td>
<td>Set by “Wrap on overflow” in the <strong>Signals</strong> section on the <strong>Data Validity Diagnostics</strong> pane of the Configuration Parameters dialog box.</td>
<td></td>
</tr>
<tr>
<td><strong>IntegerSaturationMsg</strong></td>
<td>Specify message behavior when an integer saturation occurs.</td>
<td>'none'</td>
</tr>
<tr>
<td></td>
<td>Set by “Saturate on overflow” in the <strong>Signals</strong> section on the <strong>Data Validity Diagnostics</strong> pane of the Configuration Parameters dialog box.</td>
<td></td>
</tr>
<tr>
<td><strong>InvalidFcnCallConnMsg</strong></td>
<td>Specify message behavior when an invalid function-call connection exists.</td>
<td>'warning'</td>
</tr>
<tr>
<td></td>
<td>Set by <strong>Invalid function-call connection</strong> on the <strong>Connectivity Diagnostics</strong> pane of the Configuration Parameters dialog box.</td>
<td></td>
</tr>
<tr>
<td><strong>Jacobian</strong></td>
<td>For internal use.</td>
<td></td>
</tr>
<tr>
<td><strong>LastModifiedBy</strong></td>
<td>User name of the person who last modified this model.</td>
<td>character vector</td>
</tr>
<tr>
<td><strong>LastModifiedDate</strong></td>
<td>Date when the model was last saved.</td>
<td>character vector</td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
<td>Values</td>
</tr>
<tr>
<td>-------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------------------------------------------------------------------------</td>
</tr>
<tr>
<td>LibraryLinkDisplay</td>
<td>Displays the blocks in the model that are linked or have disabled or modified links.</td>
<td>'none'</td>
</tr>
<tr>
<td>LibraryType</td>
<td>For internal use.</td>
<td></td>
</tr>
<tr>
<td>LifeSpan</td>
<td>Specify how long (in days) an application that contains blocks depending on elapsed or absolute time should be able to execute before timer overflow.</td>
<td>{'auto'}</td>
</tr>
<tr>
<td>LimitDataPoints</td>
<td>Specify that the number of data points exported to the MATLAB workspace be limited to the number specified.</td>
<td>{'on'}</td>
</tr>
<tr>
<td>LinearizationMsg</td>
<td>For internal use.</td>
<td></td>
</tr>
<tr>
<td>Lines</td>
<td>For internal use.</td>
<td></td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
<td>Values</td>
</tr>
<tr>
<td>--------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>---------------------------------------------</td>
</tr>
<tr>
<td>LineUpdate</td>
<td>Specifies when to update line routing to avoid overlapping lines and obstacles in the model. When set to 'deferred', scripts postpone line routing until they pause or complete. To perform deferred line routing within a script, you can use function <code>Simulink.BlockDiagram.routeLine</code>.</td>
<td>{'none'}</td>
</tr>
<tr>
<td>LoadExternalInput</td>
<td>Load input from workspace. Set by the <strong>Input</strong> check box on the <strong>Data Import/Export</strong> pane of the Configuration Parameters dialog box.</td>
<td>'on'</td>
</tr>
<tr>
<td>LoadInitialState</td>
<td>Load initial state from workspace. Set by the <strong>Initial state</strong> check box on the <strong>Data Import/Export</strong> pane of the Configuration Parameters dialog box.</td>
<td>'on'</td>
</tr>
<tr>
<td>Location</td>
<td>For internal use.</td>
<td></td>
</tr>
<tr>
<td>Lock</td>
<td>Lock or unlock a block library. Setting this parameter to on prevents a user from inadvertently changing a library.</td>
<td>'on'</td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
<td>Values</td>
</tr>
<tr>
<td>------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>--------------</td>
</tr>
<tr>
<td>LockLinksToLibrary</td>
<td>Lock or unlock links to a library. Setting this parameter to on prevents a user from inadvertently changing linked blocks from the Simulink Editor.</td>
<td>'on'</td>
</tr>
<tr>
<td>LoggingFileName</td>
<td>Use when you enable LoggingToFile parameter for logging to persistent storage. Specify the destination MAT-file for data logging.</td>
<td>{'out.mat'}</td>
</tr>
<tr>
<td>LoggingToFile</td>
<td>Store logging data that uses Dataset format to persistent storage (MAT-file). Using a Simulink.SimulationData.DatasetRef object to access signal logging and states logging data loads data into the model workspace incrementally. Accessing data for other kinds of logging loads all of the data at once. Use this feature when logging large amounts of data that can cause memory issues. For more information about logging large data sets, see “Log Data to Persistent Storage”.</td>
<td>'on'</td>
</tr>
<tr>
<td>MAModelExclusionFile</td>
<td>Specifies the location of the Model Advisor exclusion file. Set by the File Name field on the Model Advisor Exclusion Editor dialog box.</td>
<td>{' ' }</td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
<td>Values</td>
</tr>
<tr>
<td>-----------------------</td>
<td>-------------------------------------------------------------------------------------------------------</td>
<td>-----------------------------</td>
</tr>
<tr>
<td>MaskedZcDiagnostic</td>
<td>Control diagnostic messages related to zero-crossings that are being masked.</td>
<td>'none'</td>
</tr>
<tr>
<td>MaxConsecutiveMinStep</td>
<td>Maximum number of minimum step size violations allowed during simulation. This option appears when the solver type is Variable-step and the solver is an ode one.</td>
<td>{'1'}</td>
</tr>
<tr>
<td></td>
<td>Set by <strong>Number of consecutive min steps</strong> on the <strong>Solver</strong> pane of the Configuration Parameters dialog box.</td>
<td></td>
</tr>
<tr>
<td>MaxConsecutiveZCs</td>
<td>Maximum number of consecutive zero crossings allowed during simulation. This option appears when the solver type is Variable-step and the solver is an ode one.</td>
<td>{'1000'}</td>
</tr>
<tr>
<td></td>
<td>Set by <strong>Number of consecutive zero crossings</strong> on the <strong>Solver</strong> pane of the Configuration Parameters dialog box.</td>
<td></td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
<td>Values</td>
</tr>
<tr>
<td>--------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>-------------------------</td>
</tr>
<tr>
<td>MaxConsecutiveZCsMsg</td>
<td>Specifies diagnostic action to take when Simulink software detects the maximum number of consecutive zero crossings allowed. This option appears when the solver type is Variable-step and the solver is an ode one. Set by <strong>Consecutive zero crossings violation</strong> on the <strong>Solver</strong> section of the <strong>Diagnostics</strong> pane of the Configuration Parameters dialog box.</td>
<td>'none'</td>
</tr>
<tr>
<td>MaxDataPoints</td>
<td>Maximum number of output data points to save.</td>
<td>{'1000'}</td>
</tr>
<tr>
<td>MaxMDLFileLineLength</td>
<td>Controls the line lengths in the model file. Use this to avoid line-wrapping, which can be important for source control tools. Specifies the maximum length in bytes, which may different from the number of characters in Japanese, and is different from the number of columns when tabs are present.</td>
<td>integer — -1 (unlimited) or &gt;= 80. Default is 120.</td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
<td>Values</td>
</tr>
<tr>
<td>-----------------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>-------------------</td>
</tr>
<tr>
<td>MaxNumMinSteps</td>
<td>Maximum number of times the solver uses the minimum step size.</td>
<td>{'-1'}</td>
</tr>
<tr>
<td>MaxOrder</td>
<td>Maximum order for ode15s.</td>
<td>'1'</td>
</tr>
<tr>
<td></td>
<td>Set by Maximum order on the Solver pane of the Configuration Parameters dialog box.</td>
<td></td>
</tr>
<tr>
<td>MaxStep</td>
<td>Maximum step size.</td>
<td>{'auto'}</td>
</tr>
<tr>
<td></td>
<td>Set by Max step size on the Solver pane of the Configuration Parameters dialog box.</td>
<td></td>
</tr>
<tr>
<td>MdlSubVersion</td>
<td>For internal use</td>
<td></td>
</tr>
<tr>
<td>MergeDetectMultiDriving-BlocksExec</td>
<td>Select the diagnostic action to take when the software detects a Merge block with more than one driving block executing at the same time step.</td>
<td>{'none'}</td>
</tr>
<tr>
<td></td>
<td>Set by Detect multiple driving blocks executing at the same time step on the Configuration Parameters dialog box.</td>
<td></td>
</tr>
<tr>
<td>Metadata</td>
<td>Names and attributes of arbitrary data associated with the model. To extract this metadata structure without needing to load the model, use the method Simulink.MDLInfo.getMetadata.</td>
<td>Structure. Fields can be character vectors, numeric matrices of type &quot;double&quot;, or more structures.</td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
<td>Values</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>MinMaxOverflowArchiveData</td>
<td>For internal use</td>
<td></td>
</tr>
<tr>
<td>MinMaxOverflowArchiveMode</td>
<td>Logging type for fixed-point logging.</td>
<td>{'Overwrite'}</td>
</tr>
<tr>
<td></td>
<td>Set by <strong>Overwrite or merge model simulation results</strong> in the Fixed-Point Tool.</td>
<td></td>
</tr>
<tr>
<td>MinMaxOverflowLogging</td>
<td>Setting for fixed-point logging.</td>
<td>{'UseLocalSettings'}</td>
</tr>
<tr>
<td></td>
<td>Set by <strong>Fixed-point instrumentation mode</strong> in the Fixed-Point Tool.</td>
<td></td>
</tr>
<tr>
<td>MinStep</td>
<td>Minimum step size for the solver.</td>
<td>{'auto'}</td>
</tr>
<tr>
<td></td>
<td>Set by <strong>Min step size</strong> on the <strong>Solver</strong> pane of the Configuration Parameters dialog box.</td>
<td></td>
</tr>
<tr>
<td>MinStepSizeMsg</td>
<td>Message shown when minimum step size is violated.</td>
<td>{'warning'}</td>
</tr>
<tr>
<td></td>
<td>Set by <strong>Min step size violation</strong> on the <strong>Solver</strong> section of the <strong>Diagnostics</strong> pane of the Configuration Parameters dialog box.</td>
<td></td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
<td>Values</td>
</tr>
<tr>
<td>-----------</td>
<td>-------------</td>
<td>--------</td>
</tr>
<tr>
<td><strong>ModelBlockNormalModeVisibility</strong></td>
<td>Use with <code>set_param</code> to set normal mode visibility on for the specified Model blocks. You can set this parameter with the Model Block Normal Mode Visibility dialog box. For details, see “Specify the Instance Having Normal Mode Visibility”.</td>
<td>With <code>set_param</code>, use an array of <code>Simulink.BlockPath</code> objects or cell array of cell arrays of character vectors of paths to blocks or models. With <code>set_param</code>, an empty array specifies to use the Simulink default selection for the instance to have normal mode visibility enabled.</td>
</tr>
<tr>
<td><strong>ModelBlockNormalModeVisibilityBlockPath</strong></td>
<td>Return information about which Model blocks have normal mode visibility enabled. Use with a model that you are editing.</td>
<td>Return values indicate which Model blocks have normal mode visibility enabled. See “Simulate Multiple Referenced Model Instances in Normal Mode”.</td>
</tr>
<tr>
<td><strong>ModelBrowserVisibility</strong></td>
<td>Show the Model Browser. In the Simulink Editor, in the <strong>Modeling</strong> tab, select <strong>Environment</strong> &gt; <strong>Model Browser</strong>.</td>
<td>`'on'</td>
</tr>
<tr>
<td><strong>ModelBrowserWidth</strong></td>
<td>Width of the Model Browser pane in the model window. To display the Model Browser pane, see the <strong>ModelBrowserVisibility</strong> parameter.</td>
<td>integer — <code>{200}</code></td>
</tr>
<tr>
<td><strong>ModelDataFile</strong></td>
<td>For internal use.</td>
<td><code>{''}</code></td>
</tr>
<tr>
<td><strong>ModelDependencies</strong></td>
<td>List of model dependencies. Set by <code>Model dependencies</code> on the <strong>Model Referencing</strong> pane of the Configuration Parameters dialog box.</td>
<td><code>{''}</code></td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
<td>Values</td>
</tr>
<tr>
<td>---------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>-------------------------</td>
</tr>
</tbody>
</table>
| ModelReferenceCS-MismatchMessage | This parameter is maintained for compatibility purposes only. Do not use this parameter. You can use the Model Advisor to identify models referenced in Accelerator mode for which Simulink ignores certain configuration parameters.  

1. In the Simulink Editor, in the **Modeling** tab, click **Model Advisor**.  
2. Select **By Task**.  
3. Run the **Check diagnostic settings ignored during accelerated model reference simulation** check.  
For more information, see “Diagnostics That Are Ignored in Accelerator Mode”.  
Simulink ignores this parameter if you set it to **warning** or **error**. | {'none'} | 'warning' | 'error' |
<p>| ModelReferenceData-LoggingMessage | Message shown when there is unsupported data logging. Set by <strong>Unsupported data logging</strong> on the <strong>Model Referencing Diagnostics</strong> pane of the Configuration Parameters dialog box. | 'none' | {'warning'} | 'error' |</p>
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>ModelReferenceExtra-NoncontSigs</td>
<td>Specifies diagnostic action to take when a discrete signal appears to pass through a Model block to the input of a block with continuous states.</td>
<td>'none'</td>
</tr>
<tr>
<td></td>
<td>Set by <strong>Extraneous discrete derivative signals</strong> on the <strong>Solver</strong> section of the <strong>Diagnostics</strong> pane of the Configuration Parameters dialog box.</td>
<td></td>
</tr>
<tr>
<td>ModelReferenceIO-MismatchMessage</td>
<td>Message shown when there is a port and parameter mismatch.</td>
<td>{'none'}</td>
</tr>
<tr>
<td></td>
<td>Set by <strong>Port and parameter mismatch</strong> on the <strong>Model Referencing Diagnostics</strong> pane of the Configuration Parameters dialog box.</td>
<td></td>
</tr>
<tr>
<td>ModelReferenceIOMsg</td>
<td>Message shown when there is an invalid root Inport or Outport block connection.</td>
<td>{'none'}</td>
</tr>
<tr>
<td></td>
<td>Set by <strong>Invalid root Inport/Outport block connection</strong> on the <strong>Model Referencing Diagnostics</strong> pane of the Configuration Parameters dialog box.</td>
<td></td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
<td>Values</td>
</tr>
<tr>
<td>-------------------------------------------------------------</td>
<td>------------------------------------------------------------------------------</td>
<td>-------------------------</td>
</tr>
<tr>
<td>ModelReferenceMin-AlgLoopOccurrences</td>
<td>Toggles the minimization of algebraic loop occurrences.</td>
<td>'on'</td>
</tr>
<tr>
<td></td>
<td>Set by <strong>Minimize algebraic loop occurrences</strong> on the <strong>Model Referencing</strong> pane of the Configuration Parameters dialog box.</td>
<td></td>
</tr>
<tr>
<td>ModelReferenceNum-InstancesAllowed</td>
<td>Total number of model reference instances allowed per top model.</td>
<td>'Zero'</td>
</tr>
<tr>
<td></td>
<td>Set by <strong>Total number of instances allowed per top model</strong> on the <strong>Model Referencing</strong> pane of the Configuration Parameters dialog box.</td>
<td></td>
</tr>
<tr>
<td>ModelReferencePass-RootInputsByReference</td>
<td>Toggles the passing of scalar root inputs by value.</td>
<td>{'on'}</td>
</tr>
<tr>
<td></td>
<td>Set by “Pass fixed-size scalar root inputs by value for code generation” on the <strong>Model Referencing</strong> pane of the Configuration Parameters dialog box.</td>
<td></td>
</tr>
<tr>
<td>ModelReferenceSim-TargetVerbose</td>
<td>This parameter is deprecated and has no effect. Use AccelVerboseBuild instead.</td>
<td></td>
</tr>
<tr>
<td>ModelReferenceSymbol-NameMessage</td>
<td>For referenced models, specifies diagnostic action to take when the <strong>Maximum identifier length</strong> does not provide enough space to make global identifiers unique across models.</td>
<td>'none'</td>
</tr>
</tbody>
</table>

6-52
<table>
<thead>
<tr>
<th><strong>Parameter</strong></th>
<th><strong>Description</strong></th>
<th><strong>Values</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>ModelReferenceTargetType</td>
<td>For internal use.</td>
<td></td>
</tr>
<tr>
<td>ModelReferenceVersion-MismatchMessage</td>
<td>Message shown when there is a model block version mismatch.</td>
<td>{'none'}</td>
</tr>
<tr>
<td></td>
<td>Set by <strong>Model block version mismatch</strong> on the <strong>Model Referencing Diagnostics</strong> pane of the Configuration Parameters dialog box.</td>
<td></td>
</tr>
<tr>
<td>ModelVersion</td>
<td>Version number of model.</td>
<td>{'1.1'}</td>
</tr>
<tr>
<td>ModelVersionFormat</td>
<td>Format of model's version number.</td>
<td>{'1.%&lt;AutoIncrement: 0&gt;}'</td>
</tr>
<tr>
<td></td>
<td>Set by <strong>Model version</strong> on the <strong>History</strong> pane of the Model Properties dialog box.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>See “Model Information” for more information.</td>
<td></td>
</tr>
<tr>
<td>ModelWorkspace</td>
<td>References this model's model workspace object.</td>
<td>an instance of the Simulink.ModelWorkspace class</td>
</tr>
<tr>
<td>ModifiedByFormat</td>
<td>Format for the display of last modifier.</td>
<td>{'%&lt;Auto&gt;}'</td>
</tr>
<tr>
<td></td>
<td>Set by <strong>Last saved by</strong> on the <strong>History</strong> pane of the Model Properties dialog box.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>See “Model Information” for more information.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Can also be set by <strong>Last saved by</strong> on the <strong>Model history</strong> field on the <strong>History</strong> pane of the Model Explorer.</td>
<td></td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
<td>Values</td>
</tr>
<tr>
<td>---------------------------</td>
<td>------------------------------------------------------------------------------</td>
<td>----------------------</td>
</tr>
<tr>
<td>ModifiedComment</td>
<td>Field for user comments.</td>
<td>{''}</td>
</tr>
<tr>
<td>ModifiedDateFormat</td>
<td>Format used to generate the value of the LastModifiedDate parameter.</td>
<td>{'%&lt;Auto&gt;'}</td>
</tr>
<tr>
<td></td>
<td>Set by <strong>Last saved on</strong> on the <strong>History</strong> pane of the Model Properties dialog box.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>See “Model Information” for more information.</td>
<td></td>
</tr>
<tr>
<td>ModifiedHistory</td>
<td>Area for keeping notes about the history of the model.</td>
<td>{''}</td>
</tr>
<tr>
<td></td>
<td>Set by the <strong>Model history</strong> field on the <strong>History</strong> pane of the Model Properties dialog box.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>See “Model Information” for more information.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Can also be set by the <strong>Model history</strong> field on the <strong>History</strong> pane of the Model Explorer.</td>
<td></td>
</tr>
<tr>
<td>MultiTaskCondExecSysMsg</td>
<td>Select the diagnostic action to take if Simulink software detects a subsystem that might cause data corruption or nondeterministic behavior.</td>
<td>'none'</td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
<td>Values</td>
</tr>
<tr>
<td>---------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>---------------------------------------------</td>
</tr>
<tr>
<td>MultiTaskDSMMsg</td>
<td>Specifies diagnostic action to take when one task reads data from a Data Store Memory block to which another task writes data.</td>
<td>'none'</td>
</tr>
<tr>
<td></td>
<td>Set by Multitask data store on the Data Validity Diagnostics pane of the Configuration Parameters dialog box.</td>
<td></td>
</tr>
<tr>
<td>MultiTaskRateTransMsg</td>
<td>Specifies diagnostic action to take when an invalid rate transition takes place between two blocks operating in multitasking mode.</td>
<td>'warning'</td>
</tr>
<tr>
<td></td>
<td>Set by Multitask rate transition on the Sample Time Diagnostics pane of the Configuration Parameters dialog box.</td>
<td></td>
</tr>
<tr>
<td>Name</td>
<td>Model name.</td>
<td>character vector</td>
</tr>
<tr>
<td>NonBusSignalsTreatedAsBus</td>
<td>Detect when Simulink implicitly converts a non-bus signal to a bus signal to support connecting the signal to a block expecting a bus signal.</td>
<td>{'none'}</td>
</tr>
<tr>
<td></td>
<td>“Non-bus signals treated as bus signals” on the Diagnostics &gt; Connectivity pane of the Configuration Parameters dialog box.</td>
<td></td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
<td>Values</td>
</tr>
<tr>
<td>------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>-------------------------------</td>
</tr>
<tr>
<td>NumberNewtonIterations</td>
<td>Number of Newton's method iterations performed by the ode14x implicit fixed-step solver. Set by <strong>Number Newton's iterations</strong> on the <strong>Solver</strong> pane of the Configuration Parameters dialog box.</td>
<td>integer — {1}</td>
</tr>
<tr>
<td>NumStatesForStiffnessChecking</td>
<td>Threshold value of number of continuous states in model for stiffness calculation. If the number of continuous states in the model exceeds the NumStatesForStiffnessChecking value, auto solver uses ode15s. For more information, see “Select Solver Using Auto Solver”.</td>
<td>{''}</td>
</tr>
<tr>
<td>ObjectParameters</td>
<td>Names and attributes of model parameters.</td>
<td>structure</td>
</tr>
<tr>
<td>Open</td>
<td>For internal use.</td>
<td></td>
</tr>
<tr>
<td>OptimizeBlockIOStorage</td>
<td>Enables signal storage reuse optimization. Set by “Signal storage reuse” (Simulink Coder) on the Configuration Parameters dialog box.</td>
<td>{'on'}</td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
<td>Values</td>
</tr>
<tr>
<td>---------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>OutputOption</td>
<td>Time step output options for variable-step solvers.</td>
<td>'AdditionalOutputTimes'</td>
</tr>
<tr>
<td></td>
<td>Set by <strong>Output options</strong> parameter under <strong>Configuration Parameters</strong> &gt; <strong>Data Import/Export</strong> &gt; <strong>Additional parameters</strong>.</td>
<td></td>
</tr>
<tr>
<td>OutputSaveName</td>
<td>Workspace variable to store the model outputs.</td>
<td>{'yout'}</td>
</tr>
<tr>
<td></td>
<td>Set by the <strong>Output</strong> field on the <strong>Data Import/Export</strong> pane of the <strong>Configuration Parameters</strong> dialog box.</td>
<td></td>
</tr>
<tr>
<td>OutputTimes</td>
<td>Output times set when Set by <strong>Output options</strong> parameter under <strong>Configuration Parameters</strong> &gt; <strong>Data Import/Export</strong> &gt; <strong>Additional parameters</strong> is set to Produce additional output.</td>
<td>{'[]'}</td>
</tr>
<tr>
<td></td>
<td>Set using the <strong>Output times</strong> parameter.</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Note</strong> If the value of <strong>Output options</strong> is <strong>Produce additional output</strong> or <strong>Produce specified output only</strong>, set to a value other than the default value of '[]'.</td>
<td></td>
</tr>
<tr>
<td>PaperOrientation</td>
<td>Printing paper orientation.</td>
<td>'portrait'</td>
</tr>
<tr>
<td>PaperPosition</td>
<td>When <strong>PaperPositionMode</strong> is set to manual, this parameter determines the position and size of a diagram on paper and the size of the diagram exported as a graphic file in the units specified by <strong>PaperUnits</strong>.</td>
<td>vector — [left, bottom, width, height]</td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
<td>Values</td>
</tr>
<tr>
<td>----------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>--------------------</td>
</tr>
<tr>
<td>PaperPositionMode</td>
<td>Paper position mode.</td>
<td>{'auto'}</td>
</tr>
<tr>
<td></td>
<td>• auto</td>
<td></td>
</tr>
<tr>
<td></td>
<td>When printing, Simulink software sizes the diagram to fit the printed page.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>When exporting a diagram as a graphic image, Simulink software sizes the</td>
<td></td>
</tr>
<tr>
<td></td>
<td>exported image to be the same size as the diagram's normal size on screen.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• manual</td>
<td></td>
</tr>
<tr>
<td></td>
<td>When printing, Simulink software positions and sizes the diagram on the</td>
<td></td>
</tr>
<tr>
<td></td>
<td>page as indicated by PaperPosition.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>When exporting a diagram as a graphic image, Simulink software sizes the</td>
<td></td>
</tr>
<tr>
<td></td>
<td>exported graphic to have the height and width specified by PaperPosition.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• tiled</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Enables tiled printing.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>See “Tiled Printing” for more information.</td>
<td></td>
</tr>
<tr>
<td>PaperSize</td>
<td>Size of PaperType in PaperUnits.</td>
<td>vector — [width height] (read only)</td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
<td>Values</td>
</tr>
<tr>
<td>--------------</td>
<td>---------------</td>
<td>---------------------------------------------</td>
</tr>
<tr>
<td>PaperType</td>
<td>Printing paper type.</td>
<td>'usletter'</td>
</tr>
<tr>
<td>PaperUnits</td>
<td>Printing paper size units.</td>
<td>'normalized'</td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
<td>Values</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>--------</td>
</tr>
<tr>
<td>ParallelModelReferenceErrorOnInvalidPool</td>
<td>Specify if you want the Simulink software to perform a consistency check on the parallel pool before starting a parallel build.</td>
<td>{'on'}</td>
</tr>
<tr>
<td></td>
<td>If you set the parameter to on, the client and the remote workers must meet the following criteria for the parallel build to initiate:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• The parallel pool is open.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• The pool is spmd compatible.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• The platform is consistent between workers and client.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• The workers have a Simulink Real-Time license.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• A common compiler exists across workers and client.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>If you set the parameter to off, the software displays a warning for the first condition that fails and then performs a sequential build.</td>
<td></td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
<td>Values</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>----------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| **ParameterArgumentNames**  | List of parameters used as arguments when this model is called as a reference.  

Set by checking the **Argument** column for variables in the model workspace of the referenced model. See “Parameterize Instances of a Reusable Referenced Model”.  

| **ParameterDowncastMsg**    | Specifies diagnostic action to take when a parameter downcast occurs during simulation.  

Set by **Detect downcast** on the **Data Validity Diagnostics** pane of the Configuration Parameters dialog box.  

| **ParameterOverflowMsg**    | Specifies diagnostic action to take when a parameter overflow occurs during simulation.  

Set by **Detect overflow** on the **Data Validity Diagnostics** pane of the Configuration Parameters dialog box. | 'none' | 'warning' | {'error'} |
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>ParameterPrecision-LossMsg</td>
<td>Specifies diagnostic action to take when parameter precision loss occurs during simulation.</td>
<td>'none'</td>
</tr>
<tr>
<td></td>
<td>Set by <strong>Detect precision loss</strong> on the <strong>Data Validity Diagnostics</strong> pane of the Configuration Parameters dialog box.</td>
<td></td>
</tr>
<tr>
<td>ParameterTunabilityLossMsg</td>
<td>Specifies diagnostic action to take when a parameter cannot be tuned because it uses unsupported functions or operators.</td>
<td>'none'</td>
</tr>
<tr>
<td></td>
<td>Set by <strong>Detect loss of tunability</strong> on the <strong>Data Validity Diagnostics</strong> pane of the Configuration Parameters dialog box.</td>
<td></td>
</tr>
<tr>
<td>ParameterUnderflowMsg</td>
<td>Specifies diagnostic action to take when a parameter underflow occurs during simulation.</td>
<td>{'none'}</td>
</tr>
<tr>
<td></td>
<td>Set by <strong>Detect underflow</strong> on the <strong>Data Validity Diagnostics</strong> pane of the Configuration Parameters dialog box.</td>
<td></td>
</tr>
<tr>
<td>ParamWorkspaceSource</td>
<td>For internal use.</td>
<td></td>
</tr>
<tr>
<td>Parent</td>
<td>Name of the model or subsystem that owns this object. The value of this parameter for a model is an empty character vector.</td>
<td>{''}</td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
<td>Values</td>
</tr>
<tr>
<td>---------------------------------</td>
<td>------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>-------------------------------------------------------</td>
</tr>
<tr>
<td>Pause</td>
<td>Pause simulation callback. Set by <strong>Simulation pause function</strong> on the <strong>Callbacks</strong> pane of the Model Properties dialog box.</td>
<td>{'''}</td>
</tr>
<tr>
<td>PortDataTypeDisplayFormat</td>
<td>When you display port data types in a model by selecting <strong>Display &gt; Signals and Ports &gt; Port Data Types</strong>, choose whether to display data type aliases, base data types, or both. In the Simulink Editor, set by <strong>Display &gt; Signals and Ports &gt; Port Data Type Display Format</strong>.</td>
<td>{'AliasTypeOnly'}</td>
</tr>
<tr>
<td>PositivePriorityOrder</td>
<td>Choose the appropriate priority ordering for the real-time system targeted by this model. The Simulink Coder software uses this information to implement asynchronous data transfers. Set by <strong>Higher priority value indicates higher task priority</strong> on the <strong>Solver</strong> pane of the Configuration Parameters dialog box.</td>
<td>'on'</td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
<td>Values</td>
</tr>
<tr>
<td>---------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>--------</td>
</tr>
<tr>
<td>PostLoadFcn</td>
<td>Function invoked just after this model is loaded.</td>
<td>{' '}</td>
</tr>
<tr>
<td></td>
<td>Set by <strong>Model post-load function</strong> on the <strong>Callbacks</strong> pane of the Model Properties dialog box.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>See “Create Model Callbacks” for more information.</td>
<td></td>
</tr>
<tr>
<td>PostSaveFcn</td>
<td>Function invoked just after this model is saved to disk. Not executed for blocks inside library links.</td>
<td>{' '}</td>
</tr>
<tr>
<td></td>
<td>Set by <strong>Model post-save function</strong> on the <strong>Callbacks</strong> pane of the Model Properties dialog box.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>See “Create Model Callbacks” for more information.</td>
<td></td>
</tr>
<tr>
<td>PreLoadFcn</td>
<td>Preload callback.</td>
<td>{' '}</td>
</tr>
<tr>
<td></td>
<td>Set by <strong>Model pre-load function</strong> on the <strong>Callbacks</strong> pane of the Model Properties dialog box.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>See “Create Model Callbacks” for more information.</td>
<td></td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
<td>Values</td>
</tr>
<tr>
<td>--------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>--------</td>
</tr>
<tr>
<td>PreSaveFcn</td>
<td>Function invoked just before this model is saved to disk. Not executed for blocks inside library links, except when you are breaking the link, e.g., with <code>save_system(A, B, 'BreakUserLinks', 'on')</code>. Set by <strong>Model pre-save function</strong> on the <strong>Callbacks</strong> pane of the Model Properties dialog box. See “Create Model Callbacks” for more information.</td>
<td>'{''}</td>
</tr>
<tr>
<td>ProdBitPerChar</td>
<td>Describes the length in bits of the C <code>char</code> data type supported by the hardware board to be used by this model.</td>
<td>integer — {8}</td>
</tr>
<tr>
<td>ProdBitPerInt</td>
<td>Describes the length in bits of the C <code>int</code> data type supported by the hardware board to be used by this model.</td>
<td>integer — {32}</td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
<td>Values</td>
</tr>
<tr>
<td>---------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>-------------------------</td>
</tr>
<tr>
<td>ProdBitPerLong</td>
<td>Describes the length in bits of the C long data type supported by the hardware board to be used by this model.</td>
<td>integer — {32}</td>
</tr>
<tr>
<td></td>
<td>Set by <strong>Number of bits: long</strong> on the <strong>Hardware Implementation</strong> pane of the Configuration Parameters dialog box.</td>
<td></td>
</tr>
<tr>
<td>ProdBitPerLongLong</td>
<td>Describes the length in bits of the C long data type supported by the hardware board to be used by this model.</td>
<td>integer — {64}</td>
</tr>
<tr>
<td></td>
<td>Set by <strong>Number of bits: long long</strong> on the <strong>Hardware Implementation</strong> pane of the Configuration Parameters dialog box.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>The value of this parameter must be greater than or equal to the value of <strong>ProdBitPerLong</strong>.</td>
<td></td>
</tr>
<tr>
<td>ProdBitPerShort</td>
<td>Describes the length in bits of the C short data type supported by the hardware board to be used by this model.</td>
<td>integer — {16}</td>
</tr>
<tr>
<td></td>
<td>Set by <strong>Number of bits: short</strong> on the <strong>Hardware Implementation</strong> pane of the Configuration Parameters dialog box.</td>
<td></td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
<td>Values</td>
</tr>
<tr>
<td>----------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>---------------------------------------------</td>
</tr>
<tr>
<td>ProdEndianess</td>
<td>Describes the significance of the first byte of a data word of the hardware board to be used by this model.</td>
<td>{'Unspecified'}</td>
</tr>
<tr>
<td></td>
<td>Set by <strong>Byte ordering</strong> on the <strong>Hardware Implementation</strong> pane of the Configuration Parameters dialog box.</td>
<td></td>
</tr>
<tr>
<td>ProdEqTarget</td>
<td>Specifies that the hardware used to test the code generated from this model is the same as the production hardware or has the same characteristics.</td>
<td>{'on'}</td>
</tr>
<tr>
<td>ProdHWDeviceType</td>
<td>Predefined hardware device to specify the C language constraints for your microprocessor.</td>
<td>{'Generic-&gt;Unspecified (assume 32-bit Generic)'}</td>
</tr>
<tr>
<td></td>
<td>Set by <strong>Device vendor</strong> and <strong>Device type</strong> on the <strong>Hardware Implementation</strong> pane of the Configuration Parameters dialog box.</td>
<td></td>
</tr>
<tr>
<td>ProdIntDivRoundTo</td>
<td>Describes how the C compiler that creates production code for this model rounds the result of dividing one signed integer by another to produce a signed integer quotient.</td>
<td>'Floor'</td>
</tr>
<tr>
<td></td>
<td>Set by <strong>Signed integer division rounds to</strong> on the <strong>Hardware Implementation</strong> pane of the Configuration Parameters dialog box.</td>
<td></td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
<td>Values</td>
</tr>
<tr>
<td>---------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>---------------------------------------</td>
</tr>
<tr>
<td>ProdLargestAtomicFloat</td>
<td>Specify the largest floating-point data type that can be atomically loaded and stored on the hardware board.</td>
<td>'Float'</td>
</tr>
<tr>
<td></td>
<td>Set by <strong>Largest atomic size:</strong> <strong>floating-point</strong> on the <strong>Hardware Implementation</strong> pane of the Configuration Parameters dialog box.</td>
<td></td>
</tr>
<tr>
<td>ProdLargestAtomicInteger</td>
<td>Specify the largest integer data type that can be atomically loaded and stored on the hardware board.</td>
<td>{'Char'}</td>
</tr>
<tr>
<td></td>
<td>Set this parameter to 'LongLong' only if the production hardware supports the C <code>long long</code> data type and you have set ProdLongLongMode to 'on'.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Set by <strong>Largest atomic size:</strong> <strong>integer</strong> on the <strong>Hardware Implementation</strong> pane of the Configuration Parameters dialog box.</td>
<td></td>
</tr>
<tr>
<td>ProdLongLongMode</td>
<td>Specify that your C compiler supports the C <code>long long</code> data type. Most C99 compilers support <code>long long</code>.</td>
<td>'on'</td>
</tr>
<tr>
<td></td>
<td>Set by <strong>Support long long</strong> on the <strong>Hardware Implementation</strong> pane of the Configuration Parameters dialog box.</td>
<td></td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
<td>Values</td>
</tr>
<tr>
<td>------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>-----------------</td>
</tr>
<tr>
<td>ProdShiftRightIntArith</td>
<td>Describes whether the C compiler that creates production code for this model implements a signed integer right shift as an arithmetic right shift. Set by Shift right on a signed integer as arithmetic shift on the Hardware Implementation pane of the Configuration Parameters dialog box.</td>
<td>{'on'}</td>
</tr>
<tr>
<td>ProdWordSize</td>
<td>Describes the word length in bits of the hardware board to be used by this model. Set by Number of bits: native on the Hardware Implementation pane of the Configuration Parameters dialog box.</td>
<td>integer — {32}</td>
</tr>
<tr>
<td>Profile</td>
<td>Enables the simulation profiler for this model. In the Simulink Editor, in the Debug tab, select Performance Advisor &gt; Activate Profiler.</td>
<td>'on'</td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
<td>Values</td>
</tr>
<tr>
<td>----------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------------------------------------------------------------------------</td>
</tr>
<tr>
<td>PropagateSignalLabelsOutOfModel</td>
<td>Pass propagated signal names to output signals of Model block.</td>
<td>{'on'}</td>
</tr>
<tr>
<td></td>
<td>Set by <strong>Propagate all signal labels out of the model</strong> on the <strong>Model Referencing</strong> pane of the Configuration Parameters dialog box.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>See “Propagate all signal labels out of the model” for more information.</td>
<td></td>
</tr>
<tr>
<td>PropagateVarSize</td>
<td>Select how variable-size signals propagate through referenced models.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Set by <strong>Propagate sizes of variable-size signals</strong> on the <strong>Model Referencing</strong> pane of the Configuration Parameters dialog box.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>See “<strong>Model Configuration Parameters: Model Referencing</strong>” for more information.</td>
<td></td>
</tr>
<tr>
<td>ReadBeforeWriteMsg</td>
<td>Specifies diagnostic action to take when the model attempts to read data from a data store before it has stored data at the current time step.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Set by <strong>Detect read before write</strong> on the <strong>Data Validity Diagnostics</strong> pane of the Configuration Parameters dialog box.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

6-70
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>RecordCoverage</td>
<td>If <code>RecordCoverage</code> is set to <code>on</code>, Simulink collects and reports model coverage data during simulation. The format of this report is controlled by the values of the following parameters:</td>
<td>'on'</td>
</tr>
<tr>
<td></td>
<td><code>CovCompData</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>CovCumulativeReport</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>CovCumulativeVarName</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>CovHTMLOptions</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>CovHtmlReporting</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>CovMetricSettings</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>CovModelRefEnable</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>CovModelRefExcluded</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>CovNameIncrementing</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>CovPath</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>CovReportOnPause</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>CovSaveCumulativeToWorkspaceVar</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>CovSaveName</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>CovSaveSingleToWorkspaceVar</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td>If set to <code>off</code>, no model coverage data is collected or reported.</td>
<td></td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
<td>Values</td>
</tr>
<tr>
<td>-------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>-------------------------</td>
</tr>
<tr>
<td>Set by <strong>Entire System</strong> on the <strong>Coverage</strong> pane of the Configuration Parameters dialog box.</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Refine</strong></td>
<td>Refine factor.</td>
<td>{'1'}</td>
</tr>
<tr>
<td></td>
<td>Set by <strong>Refine factor</strong> parameter under parameter under <strong>Configuration Parameters</strong> &gt; <strong>Data Import/Export</strong> &gt; <strong>Additional parameters</strong>.</td>
<td></td>
</tr>
<tr>
<td><strong>RelTol</strong></td>
<td>Relative error tolerance.</td>
<td>{'1e-3'}</td>
</tr>
<tr>
<td></td>
<td>Set by <strong>Relative tolerance</strong> on the <strong>Solver</strong> pane of the Configuration Parameters dialog box.</td>
<td></td>
</tr>
<tr>
<td><strong>RemoveDisableFunc</strong></td>
<td>For model referencing contexts for ERT targets, remove the generated disable functions that cannot be reached from anywhere in the generated code.</td>
<td>'on'</td>
</tr>
<tr>
<td></td>
<td>Set by the <strong>“Remove disable function” (Embedded Coder)</strong> configuration parameter.</td>
<td></td>
</tr>
<tr>
<td><strong>Parameter</strong></td>
<td><strong>Description</strong></td>
<td><strong>Values</strong></td>
</tr>
<tr>
<td>--------------------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>--------------------</td>
</tr>
<tr>
<td>RemoveResetFunc</td>
<td>For model referencing contexts for ERT targets, remove the generated reset functions that cannot be reached from anywhere in the generated code. Set by the &quot;Remove reset function&quot; (Embedded Coder) configuration parameter.</td>
<td>{'on'}</td>
</tr>
<tr>
<td>ReportName</td>
<td>Name of the associated file for the Report Generator.</td>
<td>{'simulink-default.rpt'}</td>
</tr>
<tr>
<td>ReqHilite</td>
<td>Highlights all the blocks in the Simulink diagram that have requirements associated with them. In the Simulink Editor, in the Apps tab, under Verification, Validation, and Test, click Requirements Viewer. The Requirements Viewer tab appears. Click Highlight Links.</td>
<td>'on'</td>
</tr>
<tr>
<td>RequirementInfo</td>
<td>For internal use.</td>
<td></td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
<td>Values</td>
</tr>
<tr>
<td>------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------------------------------------------------------------------------</td>
</tr>
<tr>
<td>RootOutportRequire-BusObject</td>
<td>Specifies diagnostic action to take when a bus enters a root model Outport block for which a bus object has not been specified.</td>
<td>'none'</td>
</tr>
<tr>
<td>RTPrefix</td>
<td>Specifies diagnostic action to take when Simulink software encounters an object name that begins with rt.</td>
<td>'none'</td>
</tr>
<tr>
<td>RTW...</td>
<td>For information about model parameters beginning with RTW, see Configuration Parameters for Simulink Models and Parameter Reference in the Simulink Coder documentation.</td>
<td>'on'</td>
</tr>
<tr>
<td>SampleTimeAnnotations</td>
<td>In the Simulink Editor, in the Debug tab, select Information Overlays &gt; Text.</td>
<td>'on'</td>
</tr>
<tr>
<td>SampleTimeColors</td>
<td>In the Simulink Editor, in the Debug tab, select Information Overlays &gt; Colors.</td>
<td>'on'</td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
<td>Values</td>
</tr>
<tr>
<td>-------------------------</td>
<td>------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>SampleTimeConstraint</td>
<td>This option appears when the solver type is Fixed-step. Set by <strong>Periodic sample time constraint</strong> on the <strong>Solver</strong> pane of the Configuration Parameters dialog box.</td>
<td>{'Unconstrained'}</td>
</tr>
<tr>
<td>SampleTimeProperty</td>
<td>Specifies and assigns priorities to the sample times implemented by the model. This option appears when <strong>Periodic sample time constraint</strong> is set to <strong>Specified</strong>. Set by <strong>Sample time properties</strong> on the <strong>Solver</strong> pane of the Configuration Parameters dialog box.</td>
<td>Structure containing the fields SampleTime, Offset, and Priority</td>
</tr>
<tr>
<td>SavedCharacterEncoding</td>
<td>Specifies the character set used to encode this model. See the <code>slCharacterEncoding</code> command for more information.</td>
<td>character vector</td>
</tr>
<tr>
<td>SaveDefaultBlockParams</td>
<td>For internal use.</td>
<td></td>
</tr>
<tr>
<td>SavedSinceLoaded</td>
<td>Indicates whether the model has been saved since it was loaded. 'on' indicates the model has been saved.</td>
<td>'on'</td>
</tr>
<tr>
<td>SaveFinalState</td>
<td>Save final states to workspace. Set by the <strong>Final states</strong> check box on the <strong>Data Import/Export</strong> pane of the Configuration Parameters dialog box.</td>
<td>'on'</td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
<td>Values</td>
</tr>
<tr>
<td>------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>-----------------------------</td>
</tr>
<tr>
<td>SaveFormat</td>
<td>Format used to save data to the MATLAB workspace.</td>
<td>{'Dataset'}</td>
</tr>
<tr>
<td></td>
<td>Set by Format on the <strong>Data Import/Export</strong> pane of the Configuration Parameters dialog box.</td>
<td></td>
</tr>
<tr>
<td>SaveOutput</td>
<td>Save simulation output to workspace.</td>
<td>{'on'}</td>
</tr>
<tr>
<td></td>
<td>Set by the <strong>Output</strong> check box on the <strong>Data Import/Export</strong> pane of the Configuration Parameters dialog box.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Do not use a variable name that is the same as a Simulink.SimulationOutput object function name or property name.</td>
<td></td>
</tr>
<tr>
<td>SaveState</td>
<td>Save states to workspace.</td>
<td>'on'</td>
</tr>
<tr>
<td></td>
<td>Set by the <strong>States</strong> check box on the <strong>Data Import/Export</strong> pane of the Configuration Parameters dialog box.</td>
<td></td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
<td>Values</td>
</tr>
<tr>
<td>----------------------------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>-----------------------</td>
</tr>
<tr>
<td>SaveTime</td>
<td>Save simulation time to workspace.</td>
<td>{'on'}</td>
</tr>
<tr>
<td></td>
<td>Set by the Time check box on the Data Import/Export pane of the Configuration Parameters dialog box.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Do not use a variable name that is the same as a Simulink.SimulationOutput object function name or property name.</td>
<td></td>
</tr>
<tr>
<td>SaveWithDisabledLinksMsg</td>
<td>Specifies diagnostic action to take when saving a block diagram having disabled library links.</td>
<td>'none'</td>
</tr>
<tr>
<td></td>
<td>Set by Block diagram contains disabled library links on the Saving Diagnostics pane of the Configuration Parameters dialog box.</td>
<td></td>
</tr>
<tr>
<td>SaveWithParameterized‐LinksMsg</td>
<td>Specifies diagnostic action to take when saving a block diagram having parameterized library links.</td>
<td>'none'</td>
</tr>
<tr>
<td></td>
<td>Set by Block diagram contains parameterized library links on the Saving Diagnostics pane of the Configuration Parameters dialog box.</td>
<td></td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
<td>Values</td>
</tr>
<tr>
<td>-------------------------</td>
<td>------------------------------------------------------------------------------</td>
<td>------------------------------------------------------------------------</td>
</tr>
<tr>
<td>ScreenColor</td>
<td>Background color of the model window.</td>
<td>'black'</td>
</tr>
<tr>
<td>ScrollbarOffset</td>
<td>For internal use.</td>
<td></td>
</tr>
<tr>
<td>SFcnCompatibilityMsg</td>
<td>Specifies diagnostic action to take when S-function upgrades are needed.</td>
<td>{'none'}</td>
</tr>
<tr>
<td>SFExecutionAtInitializationDiag</td>
<td>Select the diagnostic action to take when Stateflow detects triggered or enabled charts that are not running at initialization.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Set by “Execute-at-Initialization disabled in presence of input events” on the Compatibility Diagnostics pane of the Configuration Parameters dialog box.</td>
<td></td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
<td>Values</td>
</tr>
<tr>
<td>---------------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>----------------------</td>
</tr>
<tr>
<td>SFInvalidInputDataAccess</td>
<td>Select the diagnostic action to take when a chart:</td>
<td>'none'</td>
</tr>
<tr>
<td>InChartInitDiag</td>
<td>• Has the ExecuteAtInitialization property set to true</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Accesses input data on a default transition or associated state entry actions, which execute at chart initialization</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Set by <strong>Invalid input data access in chart initialization</strong> on the <strong>Diagnostics &gt; Stateflow</strong> pane of the Configuration Parameters dialog box.</td>
<td></td>
</tr>
<tr>
<td>SFMachineParentedDataDiag</td>
<td>Select the diagnostic action to take when Stateflow detects machine-parented data that you can replace with chart-parented data of scope Data Store Memory.</td>
<td>'none'</td>
</tr>
<tr>
<td></td>
<td>Set by <strong>“Use of machine-parented data instead of Data Store Memory”</strong> on the <strong>Diagnostics &gt; Stateflow</strong> pane of the Configuration Parameters dialog box.</td>
<td></td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
<td>Values</td>
</tr>
<tr>
<td>------------------------------------------------</td>
<td>------------------------------------------------------------------------------</td>
<td>---------------------------------------------</td>
</tr>
<tr>
<td>SFNoUnconditionalDefaultTransitionDiag</td>
<td>Select the diagnostic action to take when a chart does not have an unconditional default transition to a state.</td>
<td>'none'</td>
</tr>
<tr>
<td></td>
<td>Set by <strong>No unconditional default transitions</strong> on the <strong>Diagnostics &gt; Stateflow</strong> pane of the Configuration Parameters dialog box.</td>
<td></td>
</tr>
<tr>
<td>SFSelfTransitionDiag</td>
<td>Select the diagnostic action to take when you can remove a self-transition on a leaf state.</td>
<td>'none'</td>
</tr>
<tr>
<td></td>
<td>Set by <strong>“Self transition on leaf state”</strong> on the <strong>Diagnostics &gt; Stateflow</strong> pane of the Configuration Parameters dialog box.</td>
<td></td>
</tr>
<tr>
<td>SFSimEcho</td>
<td>Enables output to appear in the MATLAB Command Window during simulation of a model that contains MATLAB Function blocks, Stateflow charts, or Truth Table blocks.</td>
<td>{'on'}</td>
</tr>
<tr>
<td></td>
<td>Set by <strong>Echo expressions without semicolons</strong> on the <strong>Simulation Target</strong> pane of the Configuration Parameters dialog box.</td>
<td></td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
<td>Values</td>
</tr>
<tr>
<td>----------------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------------------------------</td>
</tr>
<tr>
<td>SFTemporalDelaySmallerThanSampleTimeDiag</td>
<td>Select the diagnostic action to take when a state or transition absolute time operator uses a time value that is shorter than the sample time for the Stateflow block.</td>
<td>'none'</td>
</tr>
<tr>
<td></td>
<td>Set by “<strong>Absolute time temporal value shorter than sampling period</strong>” on the Simulation Target pane of the Configuration Parameters dialog box.</td>
<td></td>
</tr>
<tr>
<td>SFTransitionActionBeforeConditionDiag</td>
<td>Select the diagnostic action to take when a transition action is specified before a condition action in a transition path containing multiple segmented transitions.</td>
<td>'none'</td>
</tr>
<tr>
<td></td>
<td>Set by “<strong>Transition action specified before condition action</strong>” on the Diagnostics &gt; Stateflow pane of the Configuration Parameters dialog box.</td>
<td></td>
</tr>
<tr>
<td>SFTransitionOutsideNaturalParentDiag</td>
<td>Select the diagnostic action to take when a chart contains a transition that loops outside the parent state or junction.</td>
<td>'none'</td>
</tr>
<tr>
<td></td>
<td>Set by <strong>Transition outside natural parent</strong> on the Diagnostics &gt; Stateflow pane of the Configuration Parameters dialog box.</td>
<td></td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
<td>Values</td>
</tr>
<tr>
<td>-----------</td>
<td>-------------</td>
<td>--------</td>
</tr>
<tr>
<td>SFUndirectedBroadcast-EventsDiag</td>
<td>Select the diagnostic action to take when a chart contains undirected local event broadcasts.</td>
<td>'none'</td>
</tr>
<tr>
<td>SFUnexpectedBacktrackingDiag</td>
<td>Select the diagnostic action to take when a chart junction: • Does not have an unconditional transition path to a state or a terminal junction • Has multiple transition paths leading to it</td>
<td>'none'</td>
</tr>
<tr>
<td>SFUnreachableExecution-PathDiag</td>
<td>Select the diagnostic action to take when there are chart constructs not on a valid execution path.</td>
<td>'none'</td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
<td>Values</td>
</tr>
<tr>
<td>---------------------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>------------------</td>
</tr>
<tr>
<td>SFUnusedDataAndEventsDia</td>
<td>Select the diagnostic action to take for detection of unused data and events in a chart.</td>
<td>'none'</td>
</tr>
<tr>
<td></td>
<td>Set by Unused data and events on the Diagnostics &gt; Stateflow pane of the Configuration Parameters dialog box.</td>
<td></td>
</tr>
<tr>
<td>ShapePreserveControl</td>
<td>At each time step, use derivative information to improve integration accuracy.</td>
<td>'EnableAll'</td>
</tr>
<tr>
<td></td>
<td>Set by Shape preservation on the Solver pane of the Configuration Parameters dialog box.</td>
<td></td>
</tr>
<tr>
<td>ShowGrid</td>
<td>Has no effect in Simulink Editor. This parameter will be removed in a future release.</td>
<td>'on'</td>
</tr>
<tr>
<td>ShowLinearization-Annotations</td>
<td>Toggles linearization icons in the model.</td>
<td>{'on'}</td>
</tr>
<tr>
<td>ShowLineDimensions</td>
<td>Show signal dimensions on this model's block diagram.</td>
<td>'on'</td>
</tr>
<tr>
<td></td>
<td>In the Simulink Editor, in the Debug tab, select Information Overlays &gt; Signal Dimensions.</td>
<td></td>
</tr>
<tr>
<td>ShowLineDimensions-OnError</td>
<td>For internal use.</td>
<td></td>
</tr>
<tr>
<td>ShowLineWidths</td>
<td>Deprecated. Use ShowLineDimensions instead.</td>
<td></td>
</tr>
<tr>
<td>ShowLoopsOnError</td>
<td>Highlight invalid loops graphically.</td>
<td>{'on'}</td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
<td>Values</td>
</tr>
<tr>
<td>----------------------------</td>
<td>------------------------------------------------------------------------------</td>
<td>-----------------</td>
</tr>
<tr>
<td>ShowModelReference-BlockIO</td>
<td>Toggles display of I/O mismatch on block.</td>
<td>'on'</td>
</tr>
<tr>
<td></td>
<td>In the Simulink Editor, in the Debug tab, select Information Overlays &gt; Ref. Model I/O Mismatch.</td>
<td></td>
</tr>
<tr>
<td>ShowModelReference-BlockVersion</td>
<td>Toggles display of version on block.</td>
<td>'on'</td>
</tr>
<tr>
<td></td>
<td>In the Simulink Editor, in the Debug tab, select Information Overlays &gt; Ref. Model Version.</td>
<td></td>
</tr>
<tr>
<td>Shown</td>
<td>For internal use.</td>
<td></td>
</tr>
<tr>
<td>ShowPageBoundaries</td>
<td>Toggles display of page boundaries on the Simulink Editor canvas.</td>
<td>'on'</td>
</tr>
<tr>
<td></td>
<td>In the Simulink Editor, in the Simulation tab, select Print &gt; Show Page Boundaries.</td>
<td></td>
</tr>
<tr>
<td>ShowPortDataTypes</td>
<td>Show data types of ports on this model’s block diagram.</td>
<td>'on'</td>
</tr>
<tr>
<td></td>
<td>In the Simulink Editor, in the Debug tab, select Information Overlays &gt; Port Data Type.</td>
<td></td>
</tr>
<tr>
<td>ShowPortDataTypesOnError</td>
<td>For internal use.</td>
<td></td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
<td>Values</td>
</tr>
<tr>
<td>-----------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>-----------------</td>
</tr>
<tr>
<td>ShowPortUnits</td>
<td>Show units of ports, subsystem, and model block icons on the model block diagram.</td>
<td>'on'</td>
</tr>
<tr>
<td></td>
<td>In the Simulink Editor, in the <strong>Debug</strong> tab, select <strong>Information Overlays &gt; Port Units</strong>.</td>
<td></td>
</tr>
<tr>
<td>ShowStorageClass</td>
<td>Show storage classes of signals on this model's block diagram.</td>
<td>'on'</td>
</tr>
<tr>
<td></td>
<td>In the Simulink Editor, set by <strong>Storage Class</strong> on the <strong>Format &gt; Signals &amp; Ports</strong> menu.</td>
<td></td>
</tr>
<tr>
<td>ShowTestPointIcons</td>
<td>Show test point icons on this model's block diagram.</td>
<td>{'on'}</td>
</tr>
<tr>
<td></td>
<td>In the Simulink Editor, in the <strong>Debug</strong> tab, select <strong>Information OverlaysLog &amp; Testpoint</strong>.</td>
<td></td>
</tr>
<tr>
<td>ShowViewerIcons</td>
<td>Show viewer icons on this model's block diagram.</td>
<td>{'on'}</td>
</tr>
<tr>
<td></td>
<td>In the Simulink Editor, in the <strong>Debug</strong> tab, select <strong>Information OverlaysViewers</strong>.</td>
<td></td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
<td>Values</td>
</tr>
<tr>
<td>---------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>-------------------------------</td>
</tr>
<tr>
<td>SignalInfNanChecking</td>
<td>Specifies diagnostic action to take when the value of a block output is Inf or NaN at the current time step.</td>
<td>{'none'}</td>
</tr>
<tr>
<td></td>
<td>Set by <strong>Inf or NaN block output</strong> on the <strong>Data Validity Diagnostics</strong> pane of the Configuration Parameters dialog box.</td>
<td></td>
</tr>
<tr>
<td>SignalLabelMismatchMsg</td>
<td>Specifies diagnostic action to take when a signal label mismatch occurs.</td>
<td>{'none'}</td>
</tr>
<tr>
<td></td>
<td>Set by <strong>Signal label mismatch</strong> on the <strong>Connectivity Diagnostics</strong> pane of the Configuration Parameters dialog box.</td>
<td></td>
</tr>
<tr>
<td>SignalLogging</td>
<td>Globally enable signal logging for this model.</td>
<td>{'on'}</td>
</tr>
<tr>
<td></td>
<td>Set by the <strong>Signal logging</strong> check box on the <strong>Data Import/Export</strong> pane of the Configuration Parameters dialog box.</td>
<td></td>
</tr>
<tr>
<td><strong>Parameter</strong></td>
<td><strong>Description</strong></td>
<td><strong>Values</strong></td>
</tr>
<tr>
<td>----------------------------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>-------------------------</td>
</tr>
<tr>
<td>SignalLoggingName</td>
<td>Name for saving signal logging data to a workspace.</td>
<td>{'logsout'}</td>
</tr>
<tr>
<td></td>
<td>Set by the <strong>Signal logging</strong> field on the <strong>Data Import/Export</strong> pane of the Configuration Parameters dialog box.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Do not use a variable name that is the same as a Simulink.SimulationOutput object function name or property name.</td>
<td></td>
</tr>
<tr>
<td>SignalLoggingSaveFormat</td>
<td>Format for saving signal logging data.</td>
<td>{'Dataset'}</td>
</tr>
<tr>
<td></td>
<td>'ModelDataLogs' is supported for backward compatibility. However, when you open a model in R2016a or later, signal logging uses Dataset format, regardless of the setting of this parameter.</td>
<td></td>
</tr>
<tr>
<td>SignalNameFromLabel</td>
<td>Propagate signal names for Bus Creator block input signals whenever you change the name of an input signal programmatically.</td>
<td>{''}</td>
</tr>
<tr>
<td></td>
<td>Set with the <code>set_param</code> command, using either a port or line handle and a character vector specifying the signal name to propagate.</td>
<td></td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
<td>Values</td>
</tr>
<tr>
<td>----------------------------</td>
<td>------------------------------------------------------------------------------</td>
<td>---------------------------------------------</td>
</tr>
<tr>
<td>SignalRangeChecking</td>
<td>Select the diagnostic action to take when signals exceed specified minimum or maximum values.</td>
<td>{'none'}</td>
</tr>
<tr>
<td></td>
<td>Set by Simulation range checking on the Data Validity Diagnostics pane of the Configuration Parameters dialog box.</td>
<td></td>
</tr>
<tr>
<td>SignalResolutionControl</td>
<td>Control which named states and signals get resolved to Simulink signal objects.</td>
<td>'None'</td>
</tr>
<tr>
<td></td>
<td>Set by Signal resolution on the Data Validity Diagnostics pane of the Configuration Parameters dialog box.</td>
<td></td>
</tr>
<tr>
<td>SigSpecEnsureSample-TimeMsg</td>
<td>Specifies diagnostic action to take when the sample time of the source port of a signal specified by a Signal Specification block differs from the signal's destination port.</td>
<td>'none'</td>
</tr>
<tr>
<td></td>
<td>Set by Enforce sample times specified by Signal Specification blocks on the Sample Time Diagnostics pane of the Configuration Diagnostics dialog box.</td>
<td></td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
<td>Values</td>
</tr>
<tr>
<td>------------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>-------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>SimBuildMode</td>
<td>Specifies how you build the simulation target for a model that contains MATLAB Function blocks, Stateflow charts, or Truth Table blocks.</td>
<td>{'sf_incremental_build'}</td>
</tr>
<tr>
<td></td>
<td>Set by <strong>Simulation target build mode</strong> on the <strong>Simulation Target</strong> pane of the Configuration Parameters dialog box.</td>
<td></td>
</tr>
<tr>
<td>SimCompilerOptimization</td>
<td>Specifies the compiler optimization level during acceleration code generation.</td>
<td>'on'</td>
</tr>
<tr>
<td></td>
<td>Set by <strong>Compiler optimization level</strong> on the Configuration Parameters dialog box.</td>
<td></td>
</tr>
<tr>
<td>SimCtrlC</td>
<td>Enables responsiveness checks in code generated for MATLAB Function blocks.</td>
<td>{'on'}</td>
</tr>
<tr>
<td></td>
<td>Set by <strong>“Ensure responsiveness”</strong> on the <strong>Simulation Target</strong> pane of the Configuration Parameters dialog box.</td>
<td></td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
<td>Values</td>
</tr>
<tr>
<td>-----------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>--------</td>
</tr>
<tr>
<td>SimCustomHeaderCode</td>
<td>Enter code lines to appear near the top of a generated header file for a model that contains MATLAB Function blocks, Stateflow charts, or Truth Table blocks. Set by <strong>Header file</strong> on the <strong>Simulation Target &gt; Custom Code</strong> pane of the Configuration Parameters dialog box.</td>
<td>{''}</td>
</tr>
<tr>
<td>SimCustomInitializer</td>
<td>Enter code statements that execute once at the start of simulation for a model that contains MATLAB Function blocks, Stateflow charts, or Truth Table blocks. Set by <strong>Initialize function</strong> on the <strong>Simulation Target &gt; Custom Code</strong> pane of the Configuration Parameters dialog box.</td>
<td>{''}</td>
</tr>
<tr>
<td>SimCustomSourceCode</td>
<td>Enter code lines to appear near the top of a generated source code file for a model that contains MATLAB Function blocks, Stateflow charts, or Truth Table blocks. Set by <strong>Source file</strong> on the <strong>Simulation Target &gt; Custom Code</strong> pane of the Configuration Parameters dialog box.</td>
<td>{''}</td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
<td>Values</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>----------------------------------</td>
</tr>
<tr>
<td>SimCustom Terminator</td>
<td>Enter code statements that execute at the end of simulation for a model that contains MATLAB Function blocks, Stateflow charts, or Truth Table blocks.</td>
<td>{' '}</td>
</tr>
<tr>
<td></td>
<td>Set by <strong>Terminate function</strong> on the <strong>Simulation Target &gt; Custom Code</strong> pane of the Configuration Parameters dialog box.</td>
<td></td>
</tr>
<tr>
<td>SimIntegrity</td>
<td>Detects violations of memory integrity while building MATLAB Function blocks and stops simulation with a diagnostic.</td>
<td>{'on'}</td>
</tr>
<tr>
<td></td>
<td>Set by &quot;<strong>Ensure memory integrity</strong>&quot; on the <strong>Simulation Target</strong> pane of the Configuration Parameters dialog box.</td>
<td></td>
</tr>
<tr>
<td>SimParseCustom Code</td>
<td>Specify whether or not to parse the custom code and report unresolved symbols in the model.</td>
<td>{'on'}</td>
</tr>
<tr>
<td></td>
<td>Set by <strong>Parse custom code symbols</strong> on the <strong>Simulation Target &gt; Custom Code</strong> pane of the Configuration Parameters dialog box.</td>
<td></td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
<td>Values</td>
</tr>
<tr>
<td>---------------------</td>
<td>------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>-----------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>SimReservedNameArray</td>
<td>Enter the names of variables or functions in the generated code that match the names of variables or functions specified in custom code. This action prevents naming conflicts between identifiers in the generated code and in custom code for a model that contains MATLAB Function blocks, Stateflow charts, or Truth Table blocks. Set by Reserved names on the Simulation Target &gt; Symbols pane of the Configuration Parameters dialog box.</td>
<td>character vector array — {{}}</td>
</tr>
<tr>
<td>SimulationCommand</td>
<td>Executes a simulation command.</td>
<td>'start'</td>
</tr>
<tr>
<td></td>
<td><strong>Note</strong> You cannot use set_param to run a simulation in a MATLAB session that does not have a display, i.e., if you used matlab -nodisplay to start the session.</td>
<td></td>
</tr>
<tr>
<td>SimulationMode</td>
<td>Indicates whether Simulink software should run in Normal, Accelerator, Rapid Accelerator, SIL, PIL, or External mode. In the Simulink Editor, set by the Simulation &gt; Mode menu.</td>
<td>{'normal'}</td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
<td>Values</td>
</tr>
<tr>
<td>---------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>-------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>SimulationStatus</td>
<td>Indicates simulation status.</td>
<td>{'stopped'}</td>
</tr>
<tr>
<td>SimulationTime</td>
<td>Current time value for the simulation.</td>
<td>double — {0}</td>
</tr>
<tr>
<td>OperatingPointInterfaceChecksumMismatchMsg</td>
<td>Check to ensure that the interface checksum is identical to the model checksum before loading the operating point.</td>
<td>'none'</td>
</tr>
<tr>
<td>NonCurrentReleaseOperatingPointMsg</td>
<td>Check to report that the operating point was generated by a different version of Simulink. In the Diagnostics pane of the Configuration Parameters dialog box, configure the diagnostic to allow Simulink to report the message as error or warning.</td>
<td>'error'</td>
</tr>
<tr>
<td>SimUserDefines</td>
<td>Enter a space-separated list of preprocessor macro definitions to be added to the generated code for a model that contains MATLAB Function blocks, Stateflow charts, or Truth Table blocks. Set by Defines on the Simulation Target &gt; Custom Code pane of the Configuration Parameters dialog box.</td>
<td>{''}</td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
<td>Values</td>
</tr>
<tr>
<td>-----------------------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>--------</td>
</tr>
<tr>
<td>SimUserIncludeDirs</td>
<td>Enter a space-separated list of directory paths that contain files you include in the compiled target for a model that contains MATLAB Function blocks, Stateflow charts, or Truth Table blocks.</td>
<td>{''}</td>
</tr>
<tr>
<td>SimUserLibraries</td>
<td>Enter a space-separated list of static libraries that contain custom object code to link into the target for a model that contains MATLAB Function blocks, Stateflow charts, or Truth Table blocks.</td>
<td>{''}</td>
</tr>
<tr>
<td>SimUserSources</td>
<td>Enter a space-separated list of source files to compile and link into the target for a model that contains MATLAB Function blocks, Stateflow charts, or Truth Table blocks.</td>
<td>{''}</td>
</tr>
<tr>
<td><strong>Parameter</strong></td>
<td><strong>Description</strong></td>
<td><strong>Values</strong></td>
</tr>
<tr>
<td>-------------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>-----------------------------</td>
</tr>
<tr>
<td>SingleTaskRateTransMsg</td>
<td>Specifies diagnostic action to take when a rate transition takes place between two blocks operating in single-tasking mode.</td>
<td>{'none'}</td>
</tr>
<tr>
<td></td>
<td>Set by <strong>Single task rate transition</strong> on the <strong>Sample Time Diagnostics</strong> pane of the Configuration Parameters dialog box.</td>
<td></td>
</tr>
<tr>
<td>SLXCompressionType</td>
<td>Specifies compression type for new SLX files.</td>
<td>'None'</td>
</tr>
<tr>
<td></td>
<td>Set by <strong>SLX Compression</strong> on the <strong>Properties</strong> tab of the <strong>Property Inspector</strong>. See “Set SLX Compression Level”.</td>
<td></td>
</tr>
<tr>
<td>Solver</td>
<td>Solver used for the simulation.</td>
<td>'VariableStepDiscrete'</td>
</tr>
<tr>
<td>EnableMultiTasking</td>
<td>Solver mode for this model. This option appears when the solver type is <strong>Fixed-step</strong>.</td>
<td>'On'</td>
</tr>
<tr>
<td></td>
<td>Set by <strong>“Treat each discrete rate as a separate task”</strong> on the <strong>Solver</strong> pane of the Configuration Parameters dialog box.</td>
<td></td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
<td>Values</td>
</tr>
<tr>
<td>--------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>-------------------------------</td>
</tr>
<tr>
<td>SolverName</td>
<td>Solver used for the simulation. See Solver parameter for more information.</td>
<td></td>
</tr>
<tr>
<td>SolverPrmCheckMsg</td>
<td>Enables diagnostics to control when Simulink software automatically selects solver parameters. This option notifies you if:</td>
<td>{'none'}</td>
</tr>
<tr>
<td></td>
<td>• Simulink software changes a user-modified parameter to make it consistent with other model settings</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Simulink software automatically selects solver parameters for the model, such as FixedStepSize</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Set by <strong>Automatic solver parameter selection</strong> on the Solver section of the Diagnostics pane of the Configuration Parameters dialog box.</td>
<td></td>
</tr>
<tr>
<td>SolverResetMethod</td>
<td>This option appears when the solver type is Variable-step and the solver is ode15s (stiff/NDF), ode23t (Mod. stiff/Trapezoidal), or ode23tb (stiff/TR-BDF2).</td>
<td>{'Fast'}</td>
</tr>
<tr>
<td></td>
<td>Set by <strong>Solver reset method</strong> on the Solver pane of the Configuration Parameters dialog box.</td>
<td></td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
<td>Values</td>
</tr>
<tr>
<td>--------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------------------------------------------------------------------------</td>
</tr>
<tr>
<td>SolverType</td>
<td>Solver type used for the simulation.</td>
<td>{'Variable-step'}</td>
</tr>
<tr>
<td></td>
<td>Set by <strong>Type</strong> on the <strong>Solver</strong> pane of the Configuration Parameters dialog box.</td>
<td></td>
</tr>
<tr>
<td>SortedOrder</td>
<td>Show the sorted order of this model's blocks on the block diagram.</td>
<td>'on'</td>
</tr>
<tr>
<td></td>
<td>In the Simulink Editor, in the Debug tab, select <strong>Information Overlays &gt; Execution Order</strong>.</td>
<td></td>
</tr>
<tr>
<td>StartFcn</td>
<td>Start simulation callback.</td>
<td>{''}</td>
</tr>
<tr>
<td></td>
<td>Set by <strong>Simulation start function</strong> on the <strong>Callbacks</strong> pane of the Model Properties dialog box.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>See “Create Model Callbacks” for more information.</td>
<td></td>
</tr>
<tr>
<td>StartTime</td>
<td>Simulation start time.</td>
<td>{'0.0'}</td>
</tr>
<tr>
<td></td>
<td>Set by <strong>Start time</strong> on the <strong>Solver</strong> pane of the Configuration Parameters dialog box.</td>
<td></td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
<td>Values</td>
</tr>
<tr>
<td>----------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>-----------------------</td>
</tr>
<tr>
<td>StateNameClashWarn</td>
<td>Select the diagnostic action to take when a name is used for more than one state in the model.</td>
<td>'none'</td>
</tr>
<tr>
<td></td>
<td>Set by <strong>State name clash</strong> on the <strong>Solver</strong> section of the <strong>Diagnostics</strong> pane of the Configuration Parameters dialog box.</td>
<td></td>
</tr>
<tr>
<td>StateSaveName</td>
<td>State output name to be saved to workspace.</td>
<td>{'xout'}</td>
</tr>
<tr>
<td></td>
<td>Set by the <strong>States</strong> field on the <strong>Data Import/Export</strong> pane of the Configuration Parameters dialog box.</td>
<td></td>
</tr>
<tr>
<td>StatusBar</td>
<td>Has no effect in Simulink Editor. This parameter will be removed in a future release.</td>
<td>{'on'}</td>
</tr>
<tr>
<td></td>
<td>In the Simulink Editor, set by <strong>Status Bar</strong> on the <strong>View</strong> menu.</td>
<td></td>
</tr>
<tr>
<td>StiffnessThreshold</td>
<td>Threshold value to determine if the model is stiff.</td>
<td>{''}</td>
</tr>
<tr>
<td></td>
<td>A model is stiff if the stiffness exceeds the <strong>StiffnessThreshold</strong> value.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>The default value for this parameter is 1000. For more information, see “Select Solver Using Auto Solver”.</td>
<td></td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
<td>Values</td>
</tr>
<tr>
<td>---------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>-----------------------------------------</td>
</tr>
<tr>
<td>StopFcn</td>
<td>Stop simulation callback. Set by <strong>Simulation stop function</strong> on the <strong>Callbacks</strong> pane of the Model Properties dialog box. See “Create Model Callbacks” for more information.</td>
<td><code>{''}</code></td>
</tr>
<tr>
<td>StopTime</td>
<td>Simulation stop time. Set by <strong>Stop time</strong> on the <strong>Solver</strong> pane of the <strong>Configuration</strong> parameters dialog box.</td>
<td><code>{10.0}</code></td>
</tr>
<tr>
<td>StrictBusMsg</td>
<td>Specifies diagnostic action to take when Simulink software detects a signal that some blocks treat as a mux or vector, while other blocks treat the signal as a bus. To be enabled, several options in MathWorks products require this parameter be set to 'ErrorLevel1' or 'ErrorOnBusTreatedAsVector'.</td>
<td><code>{ErrorLevel1}</code></td>
</tr>
<tr>
<td></td>
<td>Set by <strong>Bus signal treated as vector</strong> on the <strong>Diagnostics Connectivity</strong> pane of the Configuration Parameters dialog box.</td>
<td><code>{WarnOnBusTreatedAsVector, ErrorOnBusTreatedAsVector}</code></td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
<td>Values</td>
</tr>
<tr>
<td>---------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>-------------------------</td>
</tr>
<tr>
<td>SupportModelReferenceSim</td>
<td>Use custom C code with Stateflow or with MATLAB Function blocks during referenced model simulation (SIM) target build for accelerator mode</td>
<td>'on'</td>
</tr>
<tr>
<td>TargetCustomCode</td>
<td>Caution Using custom C code for referenced models in accelerator mode can produce different results than if you simulate the model without using the custom code. If the custom code includes declarations of structures for buses or enumerations, the SIM target generation fails if the build results in duplicate declarations of those structures. Also, if the custom code uses a structure representing a bus or enumeration, you could get unexpected simulation results.</td>
<td></td>
</tr>
<tr>
<td>Tag</td>
<td>User-specified text that is assigned to the model's Tag parameter and saved with the model.</td>
<td>{''}</td>
</tr>
<tr>
<td>TargetBitPerChar</td>
<td>Describes the length in bits of the C char data type supported by the hardware used to test generated code.</td>
<td>integer — {8}</td>
</tr>
<tr>
<td>TargetBitPerInt</td>
<td>Describes the length in bits of the C int data type supported by the hardware used to test generated code.</td>
<td>integer — {32}</td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
<td>Values</td>
</tr>
<tr>
<td>---------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>---------------------------------</td>
</tr>
<tr>
<td>TargetBitPerLong</td>
<td>Describes the length in bits of the C long data type supported by the hardware used to test generated code.</td>
<td>integer — {32}</td>
</tr>
<tr>
<td>TargetBitPerLongLong</td>
<td>Describes the length in bits of the C long long data type supported by the hardware used to test generated code. The value of this parameter must be greater than or equal to the value of TargetBitPerLong.</td>
<td>integer — {64}</td>
</tr>
<tr>
<td>TargetBitPerShort</td>
<td>Describes the length in bits of the C short data type supported by the hardware used to test generated code.</td>
<td>integer — {16}</td>
</tr>
<tr>
<td>TargetEndianess</td>
<td>Describes the significance of the first byte of a data word of the hardware used to test generated code.</td>
<td>{'Unspecified'}</td>
</tr>
<tr>
<td>TargetHWDeviceType</td>
<td>Describes the characteristics of the hardware used to test generated code.</td>
<td>{'Generic-&gt;Unspecified (assume 32-bit Generic)'}</td>
</tr>
<tr>
<td>TargetIntDivRoundTo</td>
<td>Describes how the C compiler that creates test code for this model rounds the result of dividing one signed integer by another to produce a signed integer quotient.</td>
<td>'Floor'</td>
</tr>
<tr>
<td>TargetLargestAtomicFloat</td>
<td>Specify the largest floating-point data type that can be atomically loaded and stored on the hardware used to test code.</td>
<td>'Float'</td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
<td>Values</td>
</tr>
<tr>
<td>------------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>-------------------------------</td>
</tr>
<tr>
<td>TargetLargestAtomicInteger</td>
<td>Specify the largest integer data type that can be atomically loaded and stored on the hardware used to test code.</td>
<td>{'Char'}</td>
</tr>
<tr>
<td></td>
<td>Set this parameter to 'LongLong' only if the test hardware supports the C long long data type and you have set TargetLongLongMode to 'on'.</td>
<td></td>
</tr>
<tr>
<td>TargetLongLongMode</td>
<td>Specify that your C compiler supports the C long long data type. Most C99 compilers support long long.</td>
<td>'on'</td>
</tr>
<tr>
<td>TargetShiftRightIntArith</td>
<td>Describes whether the C compiler that creates test code for this model implements a signed integer right shift as an arithmetic right shift.</td>
<td>{'on'}</td>
</tr>
<tr>
<td>TargetTypeEmulation</td>
<td>Specifies whether Simulink Coder software displays or suppresses warning messages when emulating integer sizes in rapid prototyping environments.</td>
<td>integer — {0}</td>
</tr>
<tr>
<td>WarnSuppressLevel</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TargetWordSize</td>
<td>Describes the word length in bits of the hardware used to test generated code.</td>
<td>integer — {32}</td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
<td>Values</td>
</tr>
<tr>
<td>----------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------------------------------------------------------------------------</td>
</tr>
<tr>
<td>TasksWithSamePriorityMsg</td>
<td>Specifies diagnostic action to take when tasks have equal priority.</td>
<td>'none'</td>
</tr>
<tr>
<td></td>
<td>Set by Tasks with equal priority on the Sample Time Diagnostics pane of the Configuration Parameters dialog box.</td>
<td></td>
</tr>
<tr>
<td>TiledPageScale</td>
<td>Scales the size of the tiled page relative to the model.</td>
<td>{'1'}</td>
</tr>
<tr>
<td>TiledPaperMargins</td>
<td>Controls the size of the margins associated with each tiled page. Each element in the vector represents a margin at the particular edge.</td>
<td>vector — [left, top, right, bottom]</td>
</tr>
<tr>
<td>TimeAdjustmentMsg</td>
<td>Specifies diagnostic action to take if Simulink software makes a minor adjustment to a sample hit time while running the model.</td>
<td>{'none'}</td>
</tr>
<tr>
<td></td>
<td>Set by Sample hit time adjusting on the Configuration Parameters dialog box.</td>
<td></td>
</tr>
<tr>
<td>TimeSaveName</td>
<td>Simulation time name.</td>
<td>variable — {'tout'}</td>
</tr>
<tr>
<td></td>
<td>Set by the Time field on the Data Import/Export pane of the Configuration Parameters dialog box.</td>
<td></td>
</tr>
<tr>
<td>TLC...</td>
<td>Parameters whose names begin with TLC are used for code generation. See the Simulink Coder documentation for more information.</td>
<td></td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
<td>Values</td>
</tr>
<tr>
<td>--------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>-----------------------------</td>
</tr>
<tr>
<td>TryForcingSFnDF</td>
<td>This flag is used for backward compatibility with user S-functions that were written prior to R12.</td>
<td>'on'</td>
</tr>
<tr>
<td>TunableVars</td>
<td>List of global (tunable) parameters.</td>
<td>{''}</td>
</tr>
<tr>
<td></td>
<td>Set in the Model Parameter Configuration dialog box.</td>
<td></td>
</tr>
<tr>
<td>TunableVarsStorageClass</td>
<td>List of storage classes for their respective tunable parameters.</td>
<td>{''}</td>
</tr>
<tr>
<td></td>
<td>Set in the Model Parameter Configuration dialog box.</td>
<td></td>
</tr>
<tr>
<td>TunableVarsTypeQualifier</td>
<td>List of storage type qualifiers for their respective tunable parameters.</td>
<td>{''}</td>
</tr>
<tr>
<td></td>
<td>Set in the Model Parameter Configuration dialog box.</td>
<td></td>
</tr>
<tr>
<td>Type</td>
<td>Simulink object type (read only).</td>
<td>{'block_diagram'}</td>
</tr>
<tr>
<td>UnconnectedInputMsg</td>
<td>Unconnected input ports diagnostic.</td>
<td>'none'</td>
</tr>
<tr>
<td></td>
<td>Set by <strong>Unconnected block input ports</strong> on the <strong>Connectivity Diagnostics</strong> pane of the Configuration Parameters dialog box.</td>
<td></td>
</tr>
<tr>
<td>UnconnectedLineMsg</td>
<td>Unconnected lines diagnostic.</td>
<td>'none'</td>
</tr>
<tr>
<td></td>
<td>Set by <strong>Unconnected line</strong> on the <strong>Connectivity Diagnostics</strong> pane of the Configuration Parameters dialog box.</td>
<td></td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
<td>Values</td>
</tr>
<tr>
<td>-----------------------------------</td>
<td>------------------------------------------------------------------------------------------------</td>
<td>-----------------------------</td>
</tr>
<tr>
<td>UnconnectedOutputMsg</td>
<td>Unconnected block output ports diagnostic.</td>
<td>'none'</td>
</tr>
<tr>
<td></td>
<td>Set by <strong>Unconnected block output ports</strong> on the <strong>Connectivity Diagnostics</strong> pane of the Configuration Parameters dialog box.</td>
<td></td>
</tr>
<tr>
<td>UnderSpecifiedDataTypeMsg</td>
<td>Detect usage of heuristics to assign signal data types.</td>
<td>{'none'}</td>
</tr>
<tr>
<td></td>
<td>Set by <strong>Underspecified data types</strong> on the <strong>Data Validity Diagnostics</strong> pane of the Configuration Parameters dialog box.</td>
<td></td>
</tr>
<tr>
<td>UnderspecifiedInitializationDetection</td>
<td>Select how Simulink software handles initialization of initial conditions for conditionally executed subsystems, Merge blocks, subsystem elapsed time, and Discrete-Time Integrator blocks.</td>
<td>{'classic'}</td>
</tr>
<tr>
<td></td>
<td>Set by <strong>Underspecified initialization detection</strong> on the Configuration Parameters dialog box.</td>
<td></td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
<td>Values</td>
</tr>
<tr>
<td>----------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>-------------------</td>
</tr>
<tr>
<td>UniqueDataStoreMsg</td>
<td>Specifies diagnostic action to take when the model contains multiple Data Store Memory blocks that specify the same data store name. Set by Duplicate data store names on the Data Validity Diagnostics pane of the Configuration Parameters dialog box.</td>
<td>{'none'}</td>
</tr>
<tr>
<td>UnknownTsInhSupMsg</td>
<td>Detect blocks that have not set whether they allow the model containing them to inherit a sample time. Set by Unspecified inheritability of sample time on the Diagnostics &gt; Sample Time pane of the Configuration Parameters dialog box.</td>
<td>'none'</td>
</tr>
<tr>
<td>UnnecessaryDatatype-ConvMsg</td>
<td>Detect unnecessary data type conversion blocks. Set by Unnecessary type conversions on the Type Conversion Diagnostics pane of the Configuration Parameters dialog box.</td>
<td>{'none'}</td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
<td>Values</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>UpdateHistory</td>
<td>Specifies when to prompt the user about updating the model history. Set by Prompt to update model history on the History pane of the Model Properties dialog box or Prompt to update model history on the History pane of the Model Explorer. See “Model Information” for more information.</td>
<td>{'UpdateHistoryNever'}</td>
</tr>
<tr>
<td>UpdateModelReference-Targets</td>
<td>Specify whether to rebuild simulation and Simulink Coder targets for referenced models before updating, simulating, or generating code for this model. Set by Rebuild on the Model Referencing pane of the Configuration Parameters dialog box.</td>
<td>'IfOutOfDate'</td>
</tr>
<tr>
<td>UseAnalysisPorts</td>
<td>For internal use.</td>
<td></td>
</tr>
<tr>
<td>UseDivisionForNetSlopeComputation</td>
<td>Use division to handle net slope computations when simplicity and accuracy conditions are met.</td>
<td>{'off'}</td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
<td>Values</td>
</tr>
<tr>
<td>------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>---------------------------------</td>
</tr>
<tr>
<td>VectorMatrix-ConversionMsg</td>
<td>Detect vector-to-matrix or matrix-to-vector conversions.</td>
<td>{'none'}</td>
</tr>
<tr>
<td></td>
<td>Set by Vector/matrix block input conversion on the Type Conversion Diagnostics pane of the Configuration Parameters dialog box.</td>
<td></td>
</tr>
<tr>
<td>Version</td>
<td>Simulink version you are currently running, e.g., '7.6'.</td>
<td>double (read only)</td>
</tr>
<tr>
<td></td>
<td>If you are using a service pack, the ver function returns an additional digit, e.g., 7.4.1 (R2009bSP1). To get version information without loading the block diagram into memory, see Simulink.MDLInfo.</td>
<td></td>
</tr>
<tr>
<td>VersionLoaded</td>
<td>Simulink version that last saved the model, e.g., '7.6'.</td>
<td>double (read only)</td>
</tr>
<tr>
<td></td>
<td>If you are using a service pack, the ver function returns an additional digit, e.g., 7.4.1 (R2009bSP1). See also SavedSinceLoaded.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>To get version information without loading the block diagram into memory, see Simulink.MDLInfo.</td>
<td></td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
<td>Values</td>
</tr>
<tr>
<td>--------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------------------------------------------------------------------------</td>
</tr>
<tr>
<td>WideLines</td>
<td>Draws lines that carry vector or matrix signals wider than lines that carry scalar signals. In the Simulink Editor, set by <strong>Wide Nonscalar Lines</strong> on the <strong>Display &gt; Signals &amp; Ports</strong> menu.</td>
<td>'on'</td>
</tr>
<tr>
<td>WideVectorLines</td>
<td>Deprecated. Use WideLines instead.</td>
<td></td>
</tr>
<tr>
<td>WriteAfterReadMsg</td>
<td>Specifies diagnostic action to take when the model attempts to store data in a data store after previously reading data from it in the current time step. Set by <strong>Detect write after read</strong> on the <strong>Data Validity Diagnostics</strong> pane of the Configuration Parameters dialog box.</td>
<td>{'UseLocalSettings'}</td>
</tr>
<tr>
<td>WriteAfterWriteMsg</td>
<td>Specifies diagnostic action to take when the model attempts to store data in a data store twice in succession in the current time step. Set by <strong>Detect write after write</strong> on the <strong>Data Validity Diagnostics</strong> pane of the Configuration Parameters dialog box.</td>
<td>{'UseLocalSettings'}</td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
<td>Values</td>
</tr>
<tr>
<td>--------------------</td>
<td>------------------------------------------------------------------------------</td>
<td>---------------------------------------------</td>
</tr>
<tr>
<td>ZCThreshold</td>
<td>Specifies the deadband region used during the detection of zero crossings. Signals falling within this region are defined as having crossed through zero.</td>
<td>{'auto'}</td>
</tr>
<tr>
<td></td>
<td>Set by <strong>Signal threshold</strong> on the <strong>Solver</strong> pane of the Configuration Parameters dialog box.</td>
<td></td>
</tr>
<tr>
<td>ZeroCross</td>
<td>For internal use.</td>
<td></td>
</tr>
<tr>
<td>ZeroCrossAlgorithm</td>
<td>Specifies the algorithm to detect zero crossings when you select a variable-step solver.</td>
<td>{'Nonadaptive'}</td>
</tr>
<tr>
<td></td>
<td>Set by <strong>Algorithm</strong> on the <strong>Solver</strong> pane of the Configuration Parameters dialog box.</td>
<td></td>
</tr>
<tr>
<td>ZeroCrossControl</td>
<td>Enable zero-crossing detection.</td>
<td>{'UseLocalSettings'}</td>
</tr>
<tr>
<td></td>
<td>Set by <strong>Zero-crossing control</strong> on the <strong>Solver</strong> pane of the Configuration Parameters dialog box.</td>
<td></td>
</tr>
</tbody>
</table>
### Examples of Setting Model Parameters

These examples show how to set model parameters for the `mymodel` system.

This command sets the simulation start and stop times.

```matlab
set_param('mymodel','StartTime','5','StopTime','100')
```

This command sets the solver to `ode15s` and changes the maximum order.

```matlab
set_param('mymodel','Solver','ode15s','MaxOrder','3')
```

This command associates a `PostSaveFcn` callback.

```matlab
set_param('mymodel','PostSaveFcn','my_save_cb')
```
Common Block Properties

<table>
<thead>
<tr>
<th>In this section...</th>
</tr>
</thead>
<tbody>
<tr>
<td>“About Common Block Properties” on page 6-112</td>
</tr>
<tr>
<td>“Examples of Setting Block Properties” on page 6-131</td>
</tr>
</tbody>
</table>

About Common Block Properties

This table lists the properties common to all Simulink blocks, including block callback properties (see “Callbacks for Customized Model Behavior”). Examples of commands that change these properties follow this table (see “Examples of Setting Block Properties” on page 6-131).
## Common Block Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>AncestorBlock</td>
<td>Name of the library block that the block is linked to (for blocks with a disabled link).</td>
<td>character vector</td>
</tr>
<tr>
<td>AttributesFormatString</td>
<td>Block annotation text (corresponds to block properties).</td>
<td>character vector</td>
</tr>
<tr>
<td>BackgroundColor</td>
<td>Block background color.</td>
<td>color value</td>
</tr>
<tr>
<td></td>
<td>r, g, and b, are the red, green, blue values of the color in the range 0.0 to 1.0. If specified, the alpha value (a) is ignored.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Possible color values are 'black', 'white', 'red', 'green', 'blue', 'cyan', 'magenta', 'yellow', 'gray', 'lightBlue', 'orange', 'darkGreen'.</td>
<td></td>
</tr>
<tr>
<td>BlockDescription</td>
<td>Block description shown in the Block Properties dialog box.</td>
<td>character array</td>
</tr>
<tr>
<td>BlockDiagramType</td>
<td>Returns model if it is in an open Simulink block diagram. Returns library if it is a Simulink library.</td>
<td>'model'</td>
</tr>
<tr>
<td>BlockType</td>
<td>Block type (read-only).</td>
<td>character array</td>
</tr>
<tr>
<td>BlockKeywords</td>
<td>Associates one or more keywords with a custom library block.</td>
<td>character vector</td>
</tr>
<tr>
<td>ClipboardFcn</td>
<td>Function called when block is copied to the clipboard (Ctrl +C) or when the menu item Copy is selected.</td>
<td>function</td>
</tr>
<tr>
<td><strong>Property</strong></td>
<td><strong>Description</strong></td>
<td><strong>Values</strong></td>
</tr>
<tr>
<td>------------------</td>
<td>---------------------------------------------------------------------------------</td>
<td>---------------------------------------------------------------------------</td>
</tr>
<tr>
<td>CloseFcn</td>
<td>Function called when <code>close_system</code> is run on block.</td>
<td>function</td>
</tr>
<tr>
<td>Commented</td>
<td>Exclude block from simulation.</td>
<td>{'off'}</td>
</tr>
<tr>
<td>CompiledBusType</td>
<td>Returns whether the signal connected to a port is not a bus, is a virtual, or is a nonvirtual bus (read-only). Specify a port or line handle when using <code>get_param</code>. For example: <code>ports = get_param(gcb,'PortHandles'); feval(gcs,[],[],[],'compile'); bt = get_param(ports.Outport,'CompiledBusType')</code></td>
<td>'NOT_BUS'</td>
</tr>
</tbody>
</table>

See “Display Information About Buses”.
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>CompiledIsActive</td>
<td>Specifies whether the block status is active or not at compile time.</td>
<td>'off'</td>
</tr>
<tr>
<td></td>
<td>CompiledIsActive returns off if any one of these conditions is true at compile time:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Block is inactive path of Inline Variant.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Block is inactive choice of Variant Subsystem.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Block is commented out is within a Subsystem block that is commented out.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Block is inactive due to condition propagated from Variant Subsystem block.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CompiledIsActive returns off for inactive choices and returns on for active choices of Variant Subsystem.</td>
<td></td>
</tr>
<tr>
<td>CompiledPortComplexSignals</td>
<td>Complexity of port signals after updating diagram. You must compile the model before querying this property. For example:</td>
<td>structure array</td>
</tr>
<tr>
<td></td>
<td>vdp([],[],[],'compile');</td>
<td></td>
</tr>
<tr>
<td></td>
<td>d = get_param(gcb,'CompiledPortComplexSignals');</td>
<td></td>
</tr>
<tr>
<td></td>
<td>vdp([],[],[],'term');</td>
<td></td>
</tr>
<tr>
<td>Property</td>
<td>Description</td>
<td>Values</td>
</tr>
<tr>
<td>---------------------------</td>
<td>------------------------------------------------------------------------------</td>
<td>--------------</td>
</tr>
<tr>
<td>CompiledPortDataTypes</td>
<td>Data types of port signals after updating diagram. You must compile the model before querying this property. See CompiledPortComplexSignals.</td>
<td>structure array</td>
</tr>
<tr>
<td>CompiledPortDesignMin</td>
<td>Design minimum of port signals after updating diagram. You must compile the model before querying this property. For example:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>feval(gcs, [],[],[],'compile');</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ports = get_param(gcb,'PortHandles');</td>
<td></td>
</tr>
<tr>
<td></td>
<td>min = get_param(ports.Outport, 'CompiledPortDesignMin');</td>
<td></td>
</tr>
<tr>
<td></td>
<td>feval(model, [],[],[],'term');</td>
<td></td>
</tr>
<tr>
<td>CompiledPortDesignMax</td>
<td>Design maximum of port signals at compile time. You must compile the model before querying this property. For example:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>feval(gcs, [],[],[],'compile');</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ports = get_param(gcb,'PortHandles');</td>
<td></td>
</tr>
<tr>
<td></td>
<td>max = get_param(ports.Outport, 'CompiledPortDesignMax');</td>
<td></td>
</tr>
<tr>
<td></td>
<td>feval(model, [],[],[],'term');</td>
<td></td>
</tr>
<tr>
<td>CompiledPortDimensions</td>
<td>Dimensions of port signals after updating diagram. You must compile the model before querying this property. For details, see “Get Compiled Port Dimensions”.</td>
<td>numeric array</td>
</tr>
<tr>
<td>Property</td>
<td>Description</td>
<td>Values</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>-----------------------------</td>
</tr>
<tr>
<td>CompiledPortDimensionsMode</td>
<td>Indication whether the port signal has a variable size (after updating diagram). You must compile the model before querying this property. See “Programmatically Determine Whether Signal Line Has Variable Size”.</td>
<td>double number. 0 indicates the signal does not have a variable size. 1 indicates the signal has a variable size.</td>
</tr>
<tr>
<td>CompiledPortFrameData</td>
<td>Frame mode of port signals after updating diagram. You must compile the model before querying this property.</td>
<td>structure array</td>
</tr>
<tr>
<td>CompiledPortWidths</td>
<td>Structure of port widths after updating diagram. You must compile the model before querying this property.</td>
<td>structure array</td>
</tr>
<tr>
<td>CompiledSampleTime</td>
<td>Block sample time after updating diagram. You must compile the model before querying this property.</td>
<td>vector [sample time, offset time] or cell {[sample time 1, offset time 1]; [sample time 2, offset time 2];....[sample time n, offset time n]}</td>
</tr>
<tr>
<td>ContinueFcn</td>
<td>Function called at the restart of a simulation (after a pause).</td>
<td>function</td>
</tr>
<tr>
<td>CopyFcn</td>
<td>Function called when block is copied. See “Block Callback Parameters” for details.</td>
<td>function</td>
</tr>
<tr>
<td>DataTypeOverrideCompiled</td>
<td>For internal use.</td>
<td></td>
</tr>
<tr>
<td>DeleteFcn</td>
<td>Function called when block is deleted. See “Block Callback Parameters” for details.</td>
<td>MATLAB expression</td>
</tr>
<tr>
<td><strong>Property</strong></td>
<td><strong>Description</strong></td>
<td><strong>Values</strong></td>
</tr>
<tr>
<td>------------------------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>-------------------------------------</td>
</tr>
<tr>
<td>DestroyFcn</td>
<td>Function called when block is destroyed. See “Block Callback Parameters” for details.</td>
<td>MATLAB expression</td>
</tr>
<tr>
<td>Description</td>
<td>Description of block. Set by the <strong>Description</strong> field in the <strong>General</strong> pane of the Block Properties dialog box.</td>
<td>text and tokens</td>
</tr>
<tr>
<td>Diagnostics</td>
<td>For internal use.</td>
<td></td>
</tr>
<tr>
<td>DialogParameters</td>
<td>List of names/attributes of block-specific parameters for an unmasked block, or mask parameters for a masked block.</td>
<td>structure</td>
</tr>
<tr>
<td>DropShadow</td>
<td>Display drop shadow.</td>
<td>{'off'}</td>
</tr>
<tr>
<td>ExtModeLoggingSupported</td>
<td>Enable a block to support uploading of signal data in external mode (for example, with a scope block).</td>
<td>{'off'}</td>
</tr>
<tr>
<td>ExtModeLoggingTrig</td>
<td>Enable a block to act as the trigger block for external mode signal uploading.</td>
<td>{'off'}</td>
</tr>
<tr>
<td>ExtModeUploadOption</td>
<td>Enable a block to upload signal data in external mode when the <strong>Select all</strong> check box on the External Signal &amp; Triggering dialog box is not selected. A value of log indicates the block uploads signals. A value of none indicates the block does not upload signals. The value monitor is currently not in use. If the <strong>Select all</strong> check box on the External Signal &amp; Triggering dialog box is selected, it overrides this parameter setting.</td>
<td>{'none'}</td>
</tr>
<tr>
<td>Property</td>
<td>Description</td>
<td>Values</td>
</tr>
<tr>
<td>------------------</td>
<td>--------------------------------------------------</td>
<td>------------------------------------------------------------------------</td>
</tr>
<tr>
<td>FontAngle</td>
<td>Font angle.</td>
<td>'normal'</td>
</tr>
<tr>
<td>FontName</td>
<td>Font name.</td>
<td>character array</td>
</tr>
<tr>
<td>FontSize</td>
<td>Font size. A value of -1 specifies that this block inherits the font size specified by the DefaultBlockFontSize model parameter.</td>
<td>real {'-1'}</td>
</tr>
<tr>
<td>FontWeight</td>
<td>Font weight.</td>
<td>'light'</td>
</tr>
<tr>
<td>ForegroundColor</td>
<td>Foreground color of block icon.</td>
<td>color value</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r, g, and b, are the red, green, blue values of the color in the range 0.0 to 1.0. The value changes if it is too similar to the canvas color (ScreenColor parameter). Use get_param to return the actual value. If specified, the alpha value (a) is ignored.</td>
</tr>
<tr>
<td>Handle</td>
<td>Block handle.</td>
<td>real</td>
</tr>
<tr>
<td>Property</td>
<td>Description</td>
<td>Values</td>
</tr>
<tr>
<td>------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>-------------------</td>
</tr>
<tr>
<td>HideAutomaticName</td>
<td>Specify whether the block name given automatically by the Simulink Editor displays in the model. To hide automatic names, use the default setting of 'on'. (The HideAutomaticNames parameter for the model must also be set to 'on'.) Set to 'off' to display the name, and also set the block ShowName parameter to 'on'. Blocks whose ShowName parameter is 'off' are hidden regardless of this setting. For more information on how the parameters interact, see “Hide or Display Block Names”.</td>
<td>{'on'}</td>
</tr>
<tr>
<td>HiliteAncestors</td>
<td>For internal use.</td>
<td></td>
</tr>
<tr>
<td>InitFcn</td>
<td>Initialization function for a block. Created on the Callbacks pane of the Model Properties dialog box. For more information, see “Create Model Callbacks”. On non-masked blocks, updating the diagram or running the simulation call this function.</td>
<td>MATLAB expression</td>
</tr>
<tr>
<td>InputSignalNames</td>
<td>Names of input signals.</td>
<td>cell array</td>
</tr>
<tr>
<td>Property</td>
<td>Description</td>
<td>Values</td>
</tr>
<tr>
<td>---------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>-------------------------</td>
</tr>
<tr>
<td>IntrinsicDialogParameters</td>
<td>List of names/attributes of block-specific parameters (regardless of whether the block is masked or unmasked). Use instead of DialogParameters if you want block-specific parameters for masked or unmasked blocks.</td>
<td>structure</td>
</tr>
<tr>
<td>IOSignalStrings</td>
<td>Block paths to objects that are connected to the Signal &amp; Scope Manager. Simulink software saves these paths when the model is saved.</td>
<td>list</td>
</tr>
<tr>
<td>IOType</td>
<td>Signal &amp; Scope Manager type. For internal use.</td>
<td></td>
</tr>
<tr>
<td>LibraryVersion</td>
<td>For a linked block, the initial value of this property is the ModelVersion of the library at the time the link was created. The value updates with increments in the model version of the library.</td>
<td>character vector — {'1.1'}</td>
</tr>
<tr>
<td>LineHandles</td>
<td>Handles of lines connected to block.</td>
<td>structure</td>
</tr>
<tr>
<td>LinkData</td>
<td>Array of details about changes to the blocks inside the link that differ between a parameterized link and its library, listing the block names and parameter values. Use [] to reset to deparameterized, e.g., set_param(gcf,'linkData',[]). See “Restore Disabled or Parameterized Links”.</td>
<td>cell array</td>
</tr>
<tr>
<td><strong>Property</strong></td>
<td><strong>Description</strong></td>
<td><strong>Values</strong></td>
</tr>
<tr>
<td>------------------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>---------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>LinkStatus</td>
<td>Link status of block. Updates out-of-date linked blocks when queried using <code>get_param</code>. See “Control Linked Block Programmatically”.</td>
<td>'none'</td>
</tr>
<tr>
<td>LoadFcn</td>
<td>Function called when block is loaded.</td>
<td>MATLAB expression</td>
</tr>
<tr>
<td>MinMaxOverflow-Logging_Compiled</td>
<td>For internal use.</td>
<td></td>
</tr>
<tr>
<td>ModelCloseFcn</td>
<td>Function called when model is closed. The <code>ModelCloseFcn</code> is called prior to the block's <code>DeleteFcn</code> and <code>DestroyFcn</code> callbacks, if either are set.</td>
<td>MATLAB expression</td>
</tr>
<tr>
<td>ModelParamTableInfo</td>
<td>For internal use.</td>
<td></td>
</tr>
<tr>
<td>MoveFcn</td>
<td>Function called when block is moved.</td>
<td>MATLAB expression</td>
</tr>
<tr>
<td>Name</td>
<td>Block name.</td>
<td>character vector</td>
</tr>
<tr>
<td>NameChangeFcn</td>
<td>Function called when block name is changed.</td>
<td>MATLAB expression</td>
</tr>
<tr>
<td>NamePlacement</td>
<td>Position of block name.</td>
<td>{'normal'}</td>
</tr>
<tr>
<td>ObjectParameters</td>
<td>Names/attributes of block's parameters.</td>
<td>structure</td>
</tr>
<tr>
<td>OpenFcn</td>
<td>Function called when this Block Parameters dialog box opens.</td>
<td>MATLAB expression</td>
</tr>
<tr>
<td>Orientation</td>
<td>Where block faces.</td>
<td>{'right'}</td>
</tr>
<tr>
<td>OutputSignalNames</td>
<td>Names of output signals.</td>
<td>cell array</td>
</tr>
<tr>
<td>Parent</td>
<td>Name of the system that owns the block.</td>
<td>character vector</td>
</tr>
<tr>
<td><strong>Property</strong></td>
<td><strong>Description</strong></td>
<td><strong>Values</strong></td>
</tr>
<tr>
<td>----------------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>--------------------</td>
</tr>
<tr>
<td>ParentCloseFcn</td>
<td>Function called when parent subsystem is closed. The <code>ParentCloseFcn</code> of blocks at the root model level is not called when the model is closed.</td>
<td>MATLAB expression</td>
</tr>
<tr>
<td>PauseFcn</td>
<td>Function called at the pause of a simulation.</td>
<td>function</td>
</tr>
<tr>
<td>Property</td>
<td>Description</td>
<td>Values</td>
</tr>
<tr>
<td>------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>--------------</td>
</tr>
<tr>
<td>PortConnectivity</td>
<td>The value of this property is an array of structures, each of which describes one of the block's input or output ports. Each port structure has the following fields:</td>
<td>structure array</td>
</tr>
<tr>
<td></td>
<td>• Type</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Specifies the port's type and/or number. The value of this field can be:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• <em>n</em>, where <em>n</em> is the number of the port for data ports</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• 'enable' if the port is an enable port</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• 'trigger' if the port is a trigger port</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• 'state' for state ports</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• 'ifaction' for action ports</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• 'LConn#' for a left connection port where # is the port's number</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• 'RConn#' for a right connection port where # is the port's number</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Position</td>
<td></td>
</tr>
<tr>
<td></td>
<td>The value of this field is a two-element vector, [<em>x</em> <em>y</em>], that specifies the port's position.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• SrcBlock</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Handle of the block connected to this port. This</td>
<td></td>
</tr>
<tr>
<td>Property</td>
<td>Description</td>
<td>Values</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
<td>--------</td>
</tr>
<tr>
<td></td>
<td>field is null for output ports and -1 for unconnected input ports. <strong>SrcBlock</strong> property is a valid source handle for Variant Subsystem blocks.</td>
<td></td>
</tr>
<tr>
<td>• <strong>SrcPort</strong></td>
<td>Number of the port connected to this port, starting at zero. This field is null for both output ports and unconnected input ports.</td>
<td></td>
</tr>
<tr>
<td>• <strong>DstBlock</strong></td>
<td>Handle of the block to which this port is connected. This field is null for input ports and contains a 1-by-0 empty matrix for unconnected output ports.</td>
<td></td>
</tr>
<tr>
<td>• <strong>DstPort</strong></td>
<td>Number of the port to which this port is connected, starting at zero. This field is null for input ports and contains a 1-by-0 empty matrix for unconnected output ports.</td>
<td></td>
</tr>
<tr>
<td>Property</td>
<td>Description</td>
<td>Values</td>
</tr>
<tr>
<td>--------------</td>
<td>------------------------------------------------------------------------------</td>
<td>-------------------</td>
</tr>
<tr>
<td>PortHandles</td>
<td>The value of this property is a structure that specifies the handles of the block's ports. The structure has the following fields:</td>
<td></td>
</tr>
</tbody>
</table>
|              | • **Inport**  
|              | Handles of the block's input ports.                                          | structure array   |
|              | • **Outport**  
|              | Handles of the block's output ports.                                         |                   |
|              | • **Enable**   
|              | Handle of the block's enable port.                                           |                   |
|              | • **Trigger**  
|              | Handle of the block's trigger port.                                          |                   |
|              | • **State**    
|              | Handle of the block's state port.                                            |                   |
|              | • **LConn**    
|              | Handles of the block's left connection ports (for blocks that support Physical Modeling tools). |                   |
|              | • **RConn**    
<p>|              | Handles of the block's right connection ports (for blocks that support Physical Modeling tools). |                   |</p>
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Ifaction</td>
<td>Handle of the block's action port.</td>
<td></td>
</tr>
<tr>
<td>• Reset</td>
<td>Handle of the block's reset port.</td>
<td></td>
</tr>
<tr>
<td>PortRotationType</td>
<td>Type of port rotation used by this block (read-only).</td>
<td>'default'</td>
</tr>
</tbody>
</table>
| Ports             | A vector that specifies the number of each kind of port this block has. The order of the vector's elements corresponds to the following port types:  
• Inport  
• Outport  
• Enable  
• Trigger  
• State  
• LConn  
• RConn  
• Ifaction  
• Reset  | vector                                                                                                                                     |                      |
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Position</td>
<td>Position of block in model window.</td>
<td>vector of coordinates, in pixels: [left top right bottom]</td>
</tr>
<tr>
<td></td>
<td>To help with block alignment, the position you set can differ from the actual block position by a few pixels. Use <code>get_param</code> to return the actual position.</td>
<td>The origin is the upper-left corner of the Simulink Editor canvas before any canvas resizing. Supported coordinates are between -1073740824 and 1073740823, inclusive. Positive values are to the right of and down from the origin. Negative values are to the left of and up from the origin.</td>
</tr>
<tr>
<td>PostSaveFcn</td>
<td>Function called after the block is saved.</td>
<td>MATLAB expression</td>
</tr>
<tr>
<td>PreCopyFcn</td>
<td>Function called before the block is copied.</td>
<td>MATLAB expression</td>
</tr>
<tr>
<td>PreDeleteFcn</td>
<td>Function called before the block is deleted.</td>
<td>MATLAB expression</td>
</tr>
<tr>
<td>PreSaveFcn</td>
<td>Function called before the block is saved.</td>
<td>MATLAB expression</td>
</tr>
<tr>
<td>Priority</td>
<td>Specifies the block's order of execution relative to other blocks in the same model. Set by the <code>Priority</code> field on the <code>General</code> pane of the Block Properties dialog box.</td>
<td>character vector <code>{ '' }</code></td>
</tr>
<tr>
<td>ReferenceBlock</td>
<td>Name of the library block to which this block links.</td>
<td>character vector <code>{ '' }</code></td>
</tr>
<tr>
<td>RequirementInfo</td>
<td>For internal use.</td>
<td></td>
</tr>
<tr>
<td><strong>Property</strong></td>
<td><strong>Description</strong></td>
<td><strong>Values</strong></td>
</tr>
<tr>
<td>--------------</td>
<td>----------------</td>
<td>-----------</td>
</tr>
<tr>
<td>RTWData</td>
<td>User specified data, used by Simulink Coder software. Intended only for use with user written S-functions. See the section “S-Function RTWdata” (Simulink Coder) for details.</td>
<td></td>
</tr>
<tr>
<td>SampleTime</td>
<td>Value of the sample time parameter. See “Specify Sample Time” for more details.</td>
<td>character vector</td>
</tr>
<tr>
<td>Selected</td>
<td>Status of whether or not block is selected.</td>
<td>{'on'}</td>
</tr>
<tr>
<td>ShowName</td>
<td>Display or hide block name. To display a block name given by the Simulink Editor (automatic names), set the block 'HideAutomaticName' parameter to 'off' and ShowName to 'on'. To hide an automatic block name given by the Editor, set ShowName to 'on', HideAutomaticName to 'on', and HideAutomaticNames on the model to 'on'. For more information on how the parameters interact, see “Hide or Display Block Names”.</td>
<td>{'on'}</td>
</tr>
<tr>
<td>Property</td>
<td>Description</td>
<td>Values</td>
</tr>
<tr>
<td>----------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>----------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| SignalHierarchy      | If the signal is a bus, returns the name and hierarchy of the signals in the bus (read-only). Specify a port or line handle when using `get_param`. For example:  
```
ports = get_param(gcb,'PortHandles');
feval(gcs,[],[],[],'compile');
sh = get_param(ports.Outport,'SignalHierarchy')
```
See “Display Information About Buses”. | values reflect the structure of the signal that you specify |
| StartFcn             | Function called at the start of a simulation.                               | MATLAB expression                                                   |
| StatePerturbation-ForJacobian | State perturbation size to use during linearization. See “Change Perturbation Level of Blocks Perturbed During Linearization” (Simulink Control Design) for details. | character vector                                                     |
| StaticLinkStatus     | Link status of block. Does not update out-of-date linked blocks when queried using `get_param`. See also `LinkStatus`. | 'none' | 'resolved' | 'unresolved' | 'implicit' | 'inactive' | 'restore' | 'propagate' | 'propagateHierarchy' | 'restoreHierarchy' |
| StopFcn              | Function called at the termination of a simulation.                        | MATLAB expression                                                   |
| Tag                  | Text that appears in the block label that Simulink software generates. Set by the Tag field on the General pane of the Block Properties dialog box. | character vector {''}                                                |
| Type                 | Simulink object type (read-only).                                           | 'block'                                                              |
### Examples of Setting Block Properties

These examples illustrate how to change common block properties.

This command changes the orientation of the Gain block in the `mymodel` system so it faces the opposite direction (right to left).

```matlab
set_param('mymodel/Gain','Orientation','left')
```

This command associates an `OpenFcn` callback with the Gain block in the `mymodel` system.

```matlab
set_param('mymodel/Gain','OpenFcn','my_open_cb')
```

This command sets the `Position` property of the Gain block in the `mymodel` system. The block is 75 pixels wide by 25 pixels high.

```matlab
set_param('mymodel/Gain','Position',[50 250 125 275])
```
Block-Specific Parameters

To write scripts that create and modify models, you can use the `get_param` and `set_param` functions to query and modify the properties and parameters of a block or diagram. Use the tables to determine the programmatic name of a parameter or property in a block dialog box.

- Continuous Library Block Parameters on page 6-134
- Discontinuities Library Block Parameters on page 6-139
- Discrete Library Block Parameters on page 6-143
- Logic and Bit Operations Library Block Parameters on page 6-160
- Lookup Tables Library Block Parameters on page 6-164
- Math Operations Library Block Parameters on page 6-177
- Model Verification Library Block Parameters on page 6-202
- Model-Wide Utilities Library Block Parameters on page 6-206
- Ports & Subsystems Library Block Parameters on page 6-208
- Signal Attributes Library Block Parameters on page 6-241
- Signal Routing Library Block Parameters on page 6-249
- Sinks Library Block Parameters on page 6-259
- String Library Block Parameters
- Sources Library Block Parameters on page 6-264
- User-Defined Functions Library Block Parameters on page 6-276
- Additional Discrete Library Block Parameters on page 6-277
- Additional Math: Increment - Decrement Library Block Parameters on page 6-280

Programmatic Parameters of Blocks and Models

Programmatic parameters that describe a model are model parameters (see “Model Parameters” on page 6-2). Parameters that describe a block are block parameters. Parameters that are common to all Simulink blocks are common block parameters (see “Common Block Properties” on page 6-112). Many blocks also have unique block-specific parameters. A masked block can have mask parameters (see “Mask Parameters” on page 6-281).
The model and block properties also include callbacks, which are commands that execute when a certain model or block event occurs. These events include opening a model, simulating a model, copying a block, opening a block, etc. See “Model, Block, and Port Callbacks”.

**Tip** For block parameters that accept array values, the number of elements in the array cannot exceed what int_T can represent. This limitation applies to both simulation and Simulink Coder code generation.

The maximum number of characters that a parameter edit field can contain is 49,000.

### Block-Specific Parameters and Programmatic Equivalents

The tables list block-specific parameters for Simulink blocks. The type of the block appears in parentheses after the block name. Some Simulink blocks work as masked subsystems. The tables indicate masked blocks by adding the designation "masked subsystem" after the block type.

The type listed for nonmasked blocks is the value of the BlockType parameter (see “Common Block Properties” on page 6-112). The type listed for masked blocks is the value of the MaskType parameter (see “Mask Parameters” on page 6-281).

The **Dialog Box Prompt** column indicates the text of the prompt for the parameter in the block dialog box. The **Values** column shows the type of value required (scalar; vector; variable), the possible values (separated with a vertical line), and the default value (enclosed in braces).
Continuous Library Block Parameters

<table>
<thead>
<tr>
<th>Block (Type)/Parameter</th>
<th>Dialog Box Prompt</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Derivative (Derivative)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CoefficientInTFapproximation</td>
<td>Coefficient c in the transfer function approximation ( \frac{s}{(c*s +1)} ) used for linearization</td>
<td>{'inf'}</td>
</tr>
<tr>
<td>Integrator (Integrator)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ExternalReset</td>
<td>External reset</td>
<td>{'none'}</td>
</tr>
<tr>
<td>InitialConditionSource</td>
<td>Initial condition source</td>
<td>{'internal'}</td>
</tr>
<tr>
<td>InitialCondition</td>
<td>Initial condition</td>
<td>scalar or vector — {'0'}</td>
</tr>
<tr>
<td>LimitOutput</td>
<td>Limit output</td>
<td>{'off'}</td>
</tr>
<tr>
<td>UpperSaturationLimit</td>
<td>Upper saturation limit</td>
<td>scalar or vector — {'inf'}</td>
</tr>
<tr>
<td>LowerSaturationLimit</td>
<td>Lower saturation limit</td>
<td>scalar or vector — {'-inf'}</td>
</tr>
<tr>
<td>ShowSaturationPort</td>
<td>Show saturation port</td>
<td>{'off'}</td>
</tr>
<tr>
<td>ShowStatePort</td>
<td>Show state port</td>
<td>{'off'}</td>
</tr>
<tr>
<td>AbsoluteTolerance</td>
<td>Absolute tolerance</td>
<td>character vector, scalar, or vector — {'auto'}</td>
</tr>
<tr>
<td>IgnoreLimit</td>
<td>Ignore limit and reset when linearizing</td>
<td>{'off'}</td>
</tr>
<tr>
<td>ZeroCross</td>
<td>Enable zero–crossing detection</td>
<td>'off'</td>
</tr>
<tr>
<td>ContinuousStateAttributes</td>
<td>State Name</td>
<td>{''}</td>
</tr>
<tr>
<td>WrapState</td>
<td>Wrap state</td>
<td>{'off'}</td>
</tr>
<tr>
<td>WrappedStateUpperValue</td>
<td>Upper value of wrapped state</td>
<td>scalar or vector — {'pi'}</td>
</tr>
<tr>
<td>WrappedStateLowerValue</td>
<td>Lower value of wrapped state</td>
<td>scalar or vector — {'-pi'}</td>
</tr>
<tr>
<td>Second-Order Integrator</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ICSourceX</td>
<td>Initial condition source x</td>
<td>{'internal'}</td>
</tr>
<tr>
<td>Block (Type)/Parameter</td>
<td>Dialog Box Prompt</td>
<td>Values</td>
</tr>
<tr>
<td>------------------------</td>
<td>-------------------</td>
<td>--------</td>
</tr>
<tr>
<td>ICX</td>
<td>Initial condition x</td>
<td>scalar or vector — {'0}'</td>
</tr>
<tr>
<td>LimitX</td>
<td>Limit x</td>
<td>{'off'}</td>
</tr>
<tr>
<td>UpperLimitX</td>
<td>Upper limit x</td>
<td>scalar or vector — {'inf}'</td>
</tr>
<tr>
<td>LowerLimitX</td>
<td>Lower limit x</td>
<td>scalar or vector — {'-inf}'</td>
</tr>
<tr>
<td>WrapStateX</td>
<td>Enable wrapping of x</td>
<td>{'off'}</td>
</tr>
<tr>
<td>WrappedUpperValueX</td>
<td>Upper value for wrapping x</td>
<td>scalar or vector — {'pi}'</td>
</tr>
<tr>
<td>WrappedLowerValueX</td>
<td>Lower value for wrapping x</td>
<td>scalar or vector — {'-pi}'</td>
</tr>
<tr>
<td>AbsoluteToleranceX</td>
<td>Absolute tolerance x</td>
<td>character vector, scalar, or vector — {'auto'}</td>
</tr>
<tr>
<td>StateNameX</td>
<td>State name x</td>
<td>{}</td>
</tr>
<tr>
<td>ICSourceDXDT</td>
<td>Initial condition source dx/dt</td>
<td>{'internal'}</td>
</tr>
<tr>
<td>ICDXDT</td>
<td>Initial condition dx/dt</td>
<td>scalar or vector — {'0}'</td>
</tr>
<tr>
<td>LimitDXDT</td>
<td>Limit dx/dt</td>
<td>{'off'}</td>
</tr>
<tr>
<td>UpperLimitDXDT</td>
<td>Upper limit dx/dt</td>
<td>scalar or vector — {'inf}'</td>
</tr>
<tr>
<td>LowerLimitDXDT</td>
<td>Lower limit dx/dt</td>
<td>scalar or vector — {'-inf}'</td>
</tr>
<tr>
<td>AbsoluteToleranceDXDT</td>
<td>Absolute tolerance dx/dt</td>
<td>character vector, scalar, or vector — {'auto'}</td>
</tr>
<tr>
<td>StateNameDXDT</td>
<td>State name dx/dt</td>
<td>{}</td>
</tr>
<tr>
<td>ExternalReset</td>
<td>External reset</td>
<td>{'none'}</td>
</tr>
<tr>
<td>ZeroCross</td>
<td>Enable zero-crossing detection</td>
<td>{'on'}</td>
</tr>
<tr>
<td>ReinitDXDTwhenXreachesSaturation</td>
<td>Reinitialize dx/dt when x reaches saturation</td>
<td>{'off'}</td>
</tr>
<tr>
<td>IgnoreStateLimitsAndResetForLinearization</td>
<td>Ignore state limits and the reset for linearization</td>
<td>{'off'}</td>
</tr>
<tr>
<td>ShowOutput</td>
<td>Show output</td>
<td>{'both'}</td>
</tr>
</tbody>
</table>

**State-Space (StateSpace)**

<p>| A | A | matrix — {'1}' |</p>
<table>
<thead>
<tr>
<th>Block (Type)/Parameter</th>
<th>Dialog Box Prompt</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>B</td>
<td>matrix — {'1'}</td>
</tr>
<tr>
<td>C</td>
<td>C</td>
<td>matrix — {'1'}</td>
</tr>
<tr>
<td>D</td>
<td>D</td>
<td>matrix — {'1'}</td>
</tr>
<tr>
<td>InitialCondition</td>
<td>Initial conditions</td>
<td>vector — {'0'}</td>
</tr>
<tr>
<td>AbsoluteTolerance</td>
<td>Absolute tolerance</td>
<td>character vector, scalar, or vector — {'auto'}</td>
</tr>
<tr>
<td>ContinuousStateAttribut es</td>
<td>State Name</td>
<td>{''}</td>
</tr>
<tr>
<td>Transfer Fcn (TransferFcn)</td>
<td>Numerator</td>
<td>Numerator coefficients — vector or matrix — {'[1]'}</td>
</tr>
<tr>
<td>Denominator</td>
<td>Denominator coefficients</td>
<td>vector — {'[1 1]'}</td>
</tr>
<tr>
<td>AbsoluteTolerance</td>
<td>Absolute tolerance</td>
<td>character vector, scalar, or vector — {'auto'}</td>
</tr>
<tr>
<td>ContinuousStateAttribut es</td>
<td>State Name</td>
<td>{''}</td>
</tr>
<tr>
<td>Transport Delay (TransportDelay)</td>
<td>DelayTime</td>
<td>Time delay — scalar or vector — {'1'}</td>
</tr>
<tr>
<td>InitialOutput</td>
<td>Initial output</td>
<td>scalar or vector — {'0'}</td>
</tr>
<tr>
<td>BufferSize</td>
<td>Initial buffer size</td>
<td>scalar — {'1024'}</td>
</tr>
<tr>
<td>FixedBuffer</td>
<td>Use fixed buffer size</td>
<td>{'off'}</td>
</tr>
<tr>
<td>TransDelayFeedthrough</td>
<td>Direct feedthrough of input during linearization</td>
<td>{'off'}</td>
</tr>
<tr>
<td>PadeOrder</td>
<td>Pade order (for linearization)</td>
<td>{'0'}</td>
</tr>
<tr>
<td>Variable Time Delay (VariableTimeDelay)</td>
<td>VariableDelayType</td>
<td>Select delay type — 'Variable transport delay'</td>
</tr>
<tr>
<td>MaximumDelay</td>
<td>Maximum delay</td>
<td>scalar or vector — {'10'}</td>
</tr>
<tr>
<td>Block (Type)/Parameter</td>
<td>Dialog Box Prompt</td>
<td>Values</td>
</tr>
<tr>
<td>----------------------------</td>
<td>-----------------------------------------------</td>
<td>-----------------------------------------------------------------------</td>
</tr>
<tr>
<td>InitialOutput</td>
<td>Initial output</td>
<td>scalar or vector — {'0'}</td>
</tr>
<tr>
<td>MaximumPoints</td>
<td>Initial buffer size</td>
<td>scalar — {'1024'}</td>
</tr>
<tr>
<td>FixedBuffer</td>
<td>Use fixed buffer size</td>
<td>{'off'}</td>
</tr>
<tr>
<td>ZeroDelay</td>
<td>Handle zero delay</td>
<td>{'off'}</td>
</tr>
<tr>
<td>TransDelayFeedthrough</td>
<td>Direct feedthrough of input during linearization</td>
<td>{'off'}</td>
</tr>
<tr>
<td>PadeOrder</td>
<td>Pade order (for linearization)</td>
<td>{'0'}</td>
</tr>
<tr>
<td>ContinuousStateAttributes</td>
<td>State Name</td>
<td>{''}</td>
</tr>
</tbody>
</table>

**Variable Transport Delay (VariableTransportDelay)**

<table>
<thead>
<tr>
<th>VariableDelayType</th>
<th>Select delay type</th>
<th>{'Variable transport delay'}</th>
<th>'Variable time delay'</th>
</tr>
</thead>
<tbody>
<tr>
<td>MaximumDelay</td>
<td>Maximum delay</td>
<td>scalar or vector — {'10'}</td>
<td></td>
</tr>
<tr>
<td>InitialOutput</td>
<td>Initial output</td>
<td>scalar or vector — {'0'}</td>
<td></td>
</tr>
<tr>
<td>MaximumPoints</td>
<td>Initial buffer size</td>
<td>scalar — {'1024'}</td>
<td></td>
</tr>
<tr>
<td>FixedBuffer</td>
<td>Use fixed buffer size</td>
<td>{'off'}</td>
<td>'on'</td>
</tr>
<tr>
<td>TransDelayFeedthrough</td>
<td>Direct feedthrough of input during linearization</td>
<td>{'off'}</td>
<td>'on'</td>
</tr>
<tr>
<td>PadeOrder</td>
<td>Pade order (for linearization)</td>
<td>{'0'}</td>
<td></td>
</tr>
<tr>
<td>AbsoluteTolerance</td>
<td>Absolute tolerance</td>
<td>character vector, scalar, or vector — {'auto'}</td>
<td>{'-1'}</td>
</tr>
<tr>
<td>ContinuousStateAttributes</td>
<td>State Name</td>
<td>{''}</td>
<td>user-defined</td>
</tr>
</tbody>
</table>

**Zero-Pole (ZeroPole)**

<table>
<thead>
<tr>
<th>Zeros</th>
<th>Zeros</th>
<th>vector — {'[1]'}</th>
</tr>
</thead>
<tbody>
<tr>
<td>Poles</td>
<td>Poles</td>
<td>vector — {'[0 -1]'}</td>
</tr>
<tr>
<td>Gain</td>
<td>Gain</td>
<td>vector — {'[1]'}</td>
</tr>
</tbody>
</table>
### Model and Block Parameters

<table>
<thead>
<tr>
<th>Block (Type)/Parameter</th>
<th>Dialog Box Prompt</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>AbsoluteTolerance</td>
<td>Absolute tolerance</td>
<td>character vector, scalar, or vector — {'auto'}</td>
</tr>
<tr>
<td>ContinuousStateAttributes</td>
<td>State Name</td>
<td>{''}</td>
</tr>
</tbody>
</table>
### Discontinuities Library Block Parameters

<table>
<thead>
<tr>
<th>Block (Type)/Parameter</th>
<th>Dialog Box Prompt</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Backlash (Backlash)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BacklashWidth</td>
<td>Deadband width</td>
<td>scalar or vector — {'1'}</td>
</tr>
<tr>
<td>InitialOutput</td>
<td>Initial output</td>
<td>scalar or vector — {'0'}</td>
</tr>
<tr>
<td>ZeroCross</td>
<td>Enable zero-crossing detection</td>
<td>'off'</td>
</tr>
<tr>
<td>SampleTime</td>
<td>Sample time (-1 for inherited)</td>
<td>{'-1'}</td>
</tr>
<tr>
<td>Coulomb &amp; Viscous Friction (Coulombic and Viscous Friction) (masked subsystem)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>offset</td>
<td>Coulomb friction value (Offset)</td>
<td>{'[1 3 2 0]'}</td>
</tr>
<tr>
<td>gain</td>
<td>Coefficient of viscous friction (Gain)</td>
<td>{'1'}</td>
</tr>
<tr>
<td>Dead Zone (DeadZone)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LowerValue</td>
<td>Start of dead zone</td>
<td>scalar or vector — {'-0.5'}</td>
</tr>
<tr>
<td>UpperValue</td>
<td>End of dead zone</td>
<td>scalar or vector — {'0.5'}</td>
</tr>
<tr>
<td>SaturateOnIntegerOverfl ow</td>
<td>Saturate on integer overflow</td>
<td>'off'</td>
</tr>
<tr>
<td>LinearizeAsGain</td>
<td>Treat as gain when linearizing</td>
<td>'off'</td>
</tr>
<tr>
<td>ZeroCross</td>
<td>Enable zero-crossing detection</td>
<td>'off'</td>
</tr>
<tr>
<td>SampleTime</td>
<td>Sample time (-1 for inherited)</td>
<td>{'-1'}</td>
</tr>
<tr>
<td>Dead Zone Dynamic (Dead Zone Dynamic) (masked subsystem)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hit Crossing (HitCross)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HitCrossingOffset</td>
<td>Hit crossing offset</td>
<td>scalar or vector — {'0'}</td>
</tr>
<tr>
<td>HitCrossingDirection</td>
<td>Hit crossing direction</td>
<td>'rising'</td>
</tr>
<tr>
<td>ShowOutputPort</td>
<td>Show output port</td>
<td>'off'</td>
</tr>
<tr>
<td>ZeroCross</td>
<td>Enable zero-crossing detection</td>
<td>'off'</td>
</tr>
<tr>
<td>SampleTime</td>
<td>Sample time (-1 for inherited)</td>
<td>{'-1'}</td>
</tr>
<tr>
<td>Quantizer (Quantizer)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>QuantizationInterval</td>
<td>Quantization interval</td>
<td>scalar or vector — {'0.5'}</td>
</tr>
<tr>
<td>Block (Type)/Parameter</td>
<td>Dialog Box Prompt</td>
<td>Values</td>
</tr>
<tr>
<td>------------------------</td>
<td>-----------------------------------</td>
<td>--------------------------------------------</td>
</tr>
<tr>
<td>LinearizeAsGain</td>
<td>Treat as gain when linearizing</td>
<td>'off'</td>
</tr>
<tr>
<td>SampleTime</td>
<td>Sample time (-1 for inherited)</td>
<td>{'-1'}</td>
</tr>
<tr>
<td>Rate Limiter (RateLimiter)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RisingSlewLimit</td>
<td>Rising slew rate</td>
<td>{'1'}</td>
</tr>
<tr>
<td>FallingSlewLimit</td>
<td>Falling slew rate</td>
<td>{'-1'}</td>
</tr>
<tr>
<td>SampleTimeMode</td>
<td>Sample time mode</td>
<td>'continuous'</td>
</tr>
<tr>
<td>InitialCondition</td>
<td>Initial condition</td>
<td>{'0'}</td>
</tr>
<tr>
<td>LinearizeAsGain</td>
<td>Treat as gain when linearizing</td>
<td>'off'</td>
</tr>
<tr>
<td>Rate Limiter Dynamic (Rate Limiter Dynamic) (masked subsystem)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Relay (Relay)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OnSwitchValue</td>
<td>Switch on point</td>
<td>{'eps'}</td>
</tr>
<tr>
<td>OffSwitchValue</td>
<td>Switch off point</td>
<td>{'eps'}</td>
</tr>
<tr>
<td>OnOutputValue</td>
<td>Output when on</td>
<td>{'1'}</td>
</tr>
<tr>
<td>OffOutputValue</td>
<td>Output when off</td>
<td>{'0'}</td>
</tr>
<tr>
<td>ZeroCross</td>
<td>Enable zero-crossing detection</td>
<td>'off'</td>
</tr>
<tr>
<td>SampleTime</td>
<td>Sample time (-1 for inherited)</td>
<td>{'-1'}</td>
</tr>
<tr>
<td>OutMin</td>
<td>Output minimum</td>
<td>{'[]'}</td>
</tr>
<tr>
<td>OutMax</td>
<td>Output maximum</td>
<td>{'[]'}</td>
</tr>
<tr>
<td>OutDataTypeStr</td>
<td>Output data type</td>
<td>'Inherit: Inherit via back propagation'</td>
</tr>
</tbody>
</table>

6-140
<table>
<thead>
<tr>
<th>Block (Type)/Parameter</th>
<th>Dialog Box Prompt</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>LockScale</td>
<td>Lock output data type setting against changes by the fixed-point tools</td>
<td>{'off'}</td>
</tr>
<tr>
<td>Saturation (Saturate)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>UpperLimit</td>
<td>Upper limit</td>
<td>scalar or vector — {'0.5'}</td>
</tr>
<tr>
<td>LowerLimit</td>
<td>Lower limit</td>
<td>scalar or vector — {'-0.5'}</td>
</tr>
<tr>
<td>LinearizeAsGain</td>
<td>Treat as gain when linearizing</td>
<td>'off'</td>
</tr>
<tr>
<td>ZeroCross</td>
<td>Enable zero-crossing detection</td>
<td>'off'</td>
</tr>
<tr>
<td>SampleTime</td>
<td>Sample time (-1 for inherited)</td>
<td>{'-1'}</td>
</tr>
<tr>
<td>OutMin</td>
<td>Output minimum</td>
<td>{'[]'}</td>
</tr>
<tr>
<td>OutMax</td>
<td>Output maximum</td>
<td>{'[]'}</td>
</tr>
<tr>
<td>OutDataTypeStr</td>
<td>Output data type</td>
<td>'Inherit: Inherit via back propagation'</td>
</tr>
<tr>
<td>LockScale</td>
<td>Lock output data type setting against changes by the fixed-point tools</td>
<td>{'off'}</td>
</tr>
<tr>
<td>RndMeth</td>
<td>Integer rounding mode</td>
<td>'Ceiling'</td>
</tr>
<tr>
<td>Saturation Dynamic (Saturation Dynamic) (masked subsystem)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OutMin</td>
<td>Output minimum</td>
<td>{'[]'}</td>
</tr>
<tr>
<td>OutMax</td>
<td>Output maximum</td>
<td>{'[]'}</td>
</tr>
<tr>
<td>Block (Type)/Parameter</td>
<td>Dialog Box Prompt</td>
<td>Values</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>----------------------------------------</td>
<td>------------------------------------------------------------------------</td>
</tr>
<tr>
<td>OutDataTypeStr</td>
<td>Output data type</td>
<td>{'Inherit: Same as second input'}</td>
</tr>
<tr>
<td>OutputDataTypeScalingMode</td>
<td>Deprecated in R2007b</td>
<td></td>
</tr>
<tr>
<td>OutDataType</td>
<td>Deprecated in R2007b</td>
<td></td>
</tr>
<tr>
<td>OutScaling</td>
<td>Deprecated in R2007b</td>
<td></td>
</tr>
<tr>
<td>LockScale</td>
<td>Lock output data type setting against changes by the fixed-point tools</td>
<td>{'off'}</td>
</tr>
<tr>
<td>RndMeth</td>
<td>Integer rounding mode</td>
<td>'Ceiling'</td>
</tr>
<tr>
<td>DoSatur</td>
<td>Saturate on integer overflow</td>
<td>{'off'}</td>
</tr>
<tr>
<td>Wrap To Zero (Wrap To Zero)</td>
<td>(masked subsystem)</td>
<td></td>
</tr>
<tr>
<td>Threshold</td>
<td>Threshold</td>
<td>{'255'}</td>
</tr>
</tbody>
</table>
## Discrete Library Block Parameters

<table>
<thead>
<tr>
<th>Block (Type)/Parameter</th>
<th>Dialog Box Prompt</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Delay (Delay)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DelayLengthSource</td>
<td>Delay length &gt; Source</td>
<td>{'Dialog'}</td>
</tr>
<tr>
<td>DelayLength</td>
<td>Delay length &gt; Value</td>
<td>{'2'}</td>
</tr>
<tr>
<td>DelayLengthUpperLimit</td>
<td>Delay length &gt; Upper limit</td>
<td>{'100'}</td>
</tr>
<tr>
<td>InitialConditionSource</td>
<td>Initial condition &gt; Source</td>
<td>{'Dialog'}</td>
</tr>
<tr>
<td>InitialCondition</td>
<td>Initial condition &gt; Value</td>
<td>{'0.0'}</td>
</tr>
<tr>
<td>ExternalReset</td>
<td>External reset</td>
<td>{'None'}</td>
</tr>
<tr>
<td>InputProcessing</td>
<td>Input processing</td>
<td>'Columns as channels (frame based)'</td>
</tr>
<tr>
<td>UseCircularBuffer</td>
<td>Use circular buffer for state</td>
<td>{'off'}</td>
</tr>
<tr>
<td>PreventDirectFeedthrough</td>
<td>Prevent direct feedthrough</td>
<td>{'off'}</td>
</tr>
<tr>
<td>RemoveDelayLengthCheckInGeneratedCode</td>
<td>Remove delay length check in generated code</td>
<td>{'off'}</td>
</tr>
<tr>
<td>DiagnosticForDelayLength</td>
<td>Diagnostic for delay length</td>
<td>{'None'}</td>
</tr>
<tr>
<td>SampleTime</td>
<td>Sample time (-1 for inherited)</td>
<td>{'-1'}</td>
</tr>
<tr>
<td>StateName</td>
<td>State name</td>
<td>{''}</td>
</tr>
<tr>
<td>StateMustResolveToSignalObject</td>
<td>State name must resolve to Simulink signal object</td>
<td>{'off'}</td>
</tr>
<tr>
<td>StateSignalObject</td>
<td>Signal object class</td>
<td>Simulink.Signal object</td>
</tr>
<tr>
<td>Code generation storage class</td>
<td>Object of a class that is derived from Simulink.Signal</td>
<td></td>
</tr>
<tr>
<td>Block (Type)/Parameter</td>
<td>Dialog Box Prompt</td>
<td>Values</td>
</tr>
<tr>
<td>------------------------------</td>
<td>------------------------------------------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>StateStorageClass</td>
<td>Code generation storage class</td>
<td>{'Auto'}</td>
</tr>
<tr>
<td>CodeGenStateStorageType</td>
<td>Code generation storage type qualifier</td>
<td>{''}</td>
</tr>
<tr>
<td>Difference (Difference)</td>
<td>(masked subsystem)</td>
<td></td>
</tr>
<tr>
<td>ICPrevInput</td>
<td>Initial condition for previous input</td>
<td>{'0.0'}</td>
</tr>
<tr>
<td>OutMin</td>
<td>Output minimum</td>
<td>{'[]'}</td>
</tr>
<tr>
<td>OutMax</td>
<td>Output maximum</td>
<td>{'[]'}</td>
</tr>
<tr>
<td>OutDataTypeStr</td>
<td>Output data type</td>
<td>{'Inherit: Inherit via internal rule'}</td>
</tr>
<tr>
<td>OutputDataTypeScalingMode</td>
<td>Deprecated in R2007b</td>
<td></td>
</tr>
<tr>
<td>OutputDataType</td>
<td>Deprecated in R2007b</td>
<td></td>
</tr>
<tr>
<td>OutputScaling</td>
<td>Deprecated in R2007b</td>
<td></td>
</tr>
<tr>
<td>LockScale</td>
<td>Lock output data type setting against changes by the fixed-point tools</td>
<td>{'off'}</td>
</tr>
<tr>
<td>Block (Type)/Parameter</td>
<td>Dialog Box Prompt</td>
<td>Values</td>
</tr>
<tr>
<td>------------------------</td>
<td>-------------------</td>
<td>--------</td>
</tr>
<tr>
<td>RndMeth</td>
<td>Integer rounding mode</td>
<td>'Ceiling'</td>
</tr>
<tr>
<td>DoSatur</td>
<td>Saturate to max or min when overflows occur</td>
<td>{'off'}</td>
</tr>
</tbody>
</table>

**Discrete Derivative (Discrete Derivative) (masked subsystem)**

| gainval                | Gain value | {'1.0'} |
| ICPrevScaledInput     | Initial condition for previous weighted input K*u/Ts | {'0.0'} |
| OutMin                | Output minimum | {'[]'} |
| OutMax                | Output maximum | {'[]'} |
| OutDataTypeStr        | Output data type | {'Inherit: Inherit via internal rule'} | {'Inherit: Inherit via back propagation'} | 'double' | 'single' | 'int8' | 'uint8' | 'int16' | 'uint16' | 'int32' | 'uint32' | 'int64' | 'uint64' | 'fixdt(1,16,0)' | 'fixdt(1,16,2^0,0)' |

<p>| OutputDataTypeScalingMode | Deprecated in R2007b |
| OutDataType            | Deprecated in R2007b |
| OutScaling             | Deprecated in R2007b |
| LockScale              | Lock output data type setting against changes by the fixed-point tools | {'off'} | 'on' |</p>
<table>
<thead>
<tr>
<th>Block (Type)/Parameter</th>
<th>Dialog Box Prompt</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>RndMeth</td>
<td>Integer rounding mode</td>
<td>'Ceiling'</td>
</tr>
<tr>
<td>DoSatur</td>
<td>Saturate to max or min when overflows occur</td>
<td>{'off'}</td>
</tr>
</tbody>
</table>

**Discrete FIR Filter (Discrete FIR Filter)**

<table>
<thead>
<tr>
<th>Block (Type)/Parameter</th>
<th>Dialog Box Prompt</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>CoefSource</td>
<td>Coefficient source</td>
<td>{'Dialog parameters'}</td>
</tr>
<tr>
<td>FilterStructure</td>
<td>Filter structure</td>
<td>{'Direct form'}</td>
</tr>
</tbody>
</table>

**Note** You must have a DSP System Toolbox license to use a filter structure other than Direct form.

<table>
<thead>
<tr>
<th>Block (Type)/Parameter</th>
<th>Dialog Box Prompt</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coefficients</td>
<td>Coefficients</td>
<td>vector — {'[0.5 0.5]'}</td>
</tr>
<tr>
<td>InputProcessing</td>
<td>Input processing</td>
<td>'Columns as channels (frame based)'</td>
</tr>
<tr>
<td>InitialStates</td>
<td>Initial states</td>
<td>scalar or vector — {'0'}</td>
</tr>
<tr>
<td>SampleTime</td>
<td>Sample time (-1 for inherited)</td>
<td>{'-1'}</td>
</tr>
<tr>
<td>CoefMin</td>
<td>Coefficients minimum</td>
<td>{'[]'}</td>
</tr>
<tr>
<td>CoefMax</td>
<td>Coefficients maximum</td>
<td>{'[]'}</td>
</tr>
<tr>
<td>OutMin</td>
<td>Output minimum</td>
<td>{'[]'}</td>
</tr>
<tr>
<td>OutMax</td>
<td>Output maximum</td>
<td>{'[]'}</td>
</tr>
<tr>
<td>Block (Type)/Parameter</td>
<td>Dialog Box Prompt</td>
<td>Values</td>
</tr>
<tr>
<td>------------------------</td>
<td>------------------</td>
<td>--------</td>
</tr>
<tr>
<td>TapSumDataTypeStr</td>
<td>Tap sum data type</td>
<td>{'Inherit: Same as input'}</td>
</tr>
<tr>
<td>CoefDataTypeStr</td>
<td>Coefficients data type</td>
<td>{'Inherit: Same word length as input'}</td>
</tr>
<tr>
<td>ProductDataTypeStr</td>
<td>Product output data type</td>
<td>{'Inherit: Inherit via internal rule'}</td>
</tr>
<tr>
<td>AccumDataTypeStr</td>
<td>Accumulator data type</td>
<td>{'Inherit: Inherit via internal rule'}</td>
</tr>
<tr>
<td>StateDataTypeStr</td>
<td>State data type</td>
<td>'Inherit: Same as input'</td>
</tr>
</tbody>
</table>

6-147
### Model and Block Parameters

<table>
<thead>
<tr>
<th><strong>Block (Type)/Parameter</strong></th>
<th><strong>Dialog Box Prompt</strong></th>
<th><strong>Values</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>OutDataTypeStr</td>
<td>Output data type</td>
<td>'Inherit: Same as input'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{'Inherit: Same as accumulator'}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>'int8'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>'uint8'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>'int16'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>'uint16'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>'int32'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>'uint32'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>'int64'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>'uint64'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>'fixdt(1,16)'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>'fixdt(1,16,0)'</td>
</tr>
<tr>
<td>LockScale</td>
<td>Lock data type settings against changes by the fixed-point tools</td>
<td>{'off'}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>'on'</td>
</tr>
<tr>
<td>RndMeth</td>
<td>Integer rounding mode</td>
<td>'Ceiling'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>'Convergent'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{'Floor'}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>'Nearest'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>'Round'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>'Simplest'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>'Zero'</td>
</tr>
<tr>
<td>SaturateOnIntegerOverfl ow</td>
<td>Saturate on integer overflow</td>
<td>{'off'}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>'on'</td>
</tr>
</tbody>
</table>

**Discrete Filter (DiscreteFilter)**

<table>
<thead>
<tr>
<th><strong>Parameter</strong></th>
<th><strong>Dialog Box Prompt</strong></th>
<th><strong>Values</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Numerator</td>
<td>Numerator coefficients</td>
<td>vector — {'[1]'}</td>
</tr>
<tr>
<td>Denominator</td>
<td>Denominator coefficients</td>
<td>vector — {'[1 0.5]'}</td>
</tr>
<tr>
<td>IC</td>
<td>Initial states</td>
<td>{'0'}</td>
</tr>
<tr>
<td>SampleTime</td>
<td>Sample time (-1 for inherited)</td>
<td>{'1'}</td>
</tr>
<tr>
<td>a0EqualsOne</td>
<td>Optimize by skipping divide by leading denominator coefficient (a0)</td>
<td>{'off'}</td>
</tr>
<tr>
<td>NumCoefMin</td>
<td>Numerator coefficient minimum</td>
<td>{'[]'}</td>
</tr>
<tr>
<td>NumCoefMax</td>
<td>Numerator coefficient maximum</td>
<td>{'[]'}</td>
</tr>
<tr>
<td>DenCoefMin</td>
<td>Denominator coefficient minimum</td>
<td>{'[]'}</td>
</tr>
<tr>
<td>DenCoefMax</td>
<td>Denominator coefficient maximum</td>
<td>{'[]'}</td>
</tr>
<tr>
<td>OutMin</td>
<td>Output minimum</td>
<td>{'[]'}</td>
</tr>
<tr>
<td>OutMax</td>
<td>Output maximum</td>
<td>{'[]'}</td>
</tr>
<tr>
<td>Block (Type)/Parameter</td>
<td>Dialog Box Prompt</td>
<td>Values</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>------------------------------------------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>StateDataTypeStr</td>
<td>State data type</td>
<td>{'Inherit: Same as input'}</td>
</tr>
<tr>
<td>NumCoefDataTypeStr</td>
<td>Numerator coefficient data type</td>
<td>{'Inherit: Inherit via internal rule'}</td>
</tr>
<tr>
<td>DenCoefDataTypeStr</td>
<td>Denominator coefficient data type</td>
<td>{'Inherit: Inherit via internal rule'}</td>
</tr>
<tr>
<td>NumProductDataTypeStr</td>
<td>Numerator product output data type</td>
<td>{'Inherit: Inherit via internal rule'}</td>
</tr>
<tr>
<td>DenProductDataTypeStr</td>
<td>Denominator product output data type</td>
<td>{'Inherit: Inherit via internal rule'}</td>
</tr>
<tr>
<td>NumAccumDataTypeStr</td>
<td>Numerator accumulator data type</td>
<td>{'Inherit: Inherit via internal rule'}</td>
</tr>
<tr>
<td>Block (Type)/Parameter</td>
<td>Dialog Box Prompt</td>
<td>Values</td>
</tr>
<tr>
<td>------------------------------</td>
<td>--------------------------------------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>DenAccumDataTypeStr</td>
<td>Denominator accumulator data type</td>
<td>{'Inherit: Inherit via internal rule'}</td>
</tr>
<tr>
<td>OutDataTypeStr</td>
<td>Output data type</td>
<td>{'Inherit: Inherit via internal rule'}</td>
</tr>
<tr>
<td>LockScale</td>
<td>Lock data type settings against changes by the fixed-point tools</td>
<td>{'off'}</td>
</tr>
<tr>
<td>RndMeth</td>
<td>Integer rounding mode</td>
<td>'Ceiling'</td>
</tr>
<tr>
<td>SaturateOnIntegerOverflow</td>
<td>Saturate on integer overflow</td>
<td>{'off'}</td>
</tr>
<tr>
<td>StateName</td>
<td>State name</td>
<td>'{}'</td>
</tr>
<tr>
<td>StateMustResolveToSignalObject</td>
<td>State name must resolve to Simulink signal object</td>
<td>{'off'}</td>
</tr>
<tr>
<td>StateSignalObject</td>
<td>Signal object class</td>
<td>Simulink.Signal object</td>
</tr>
<tr>
<td>StateStorageClass</td>
<td>Code generation storage class</td>
<td>Object of a class that is derived from Simulink.Signal</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{'Auto'}</td>
</tr>
<tr>
<td>Block (Type)/Parameter</td>
<td>Dialog Box Prompt</td>
<td>Values</td>
</tr>
<tr>
<td>----------------------------------------</td>
<td>-----------------------------------------------------------------------------------</td>
<td>-------------------------------------</td>
</tr>
<tr>
<td>RTWStateStorageTypeQualifier</td>
<td>Code generation storage type qualifier</td>
<td>{'''}</td>
</tr>
<tr>
<td>Discrete State-Space (DiscreteStateSpace)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>A</td>
<td>matrix — {'1'}</td>
</tr>
<tr>
<td>B</td>
<td>B</td>
<td>matrix — {'1'}</td>
</tr>
<tr>
<td>C</td>
<td>C</td>
<td>matrix — {'1'}</td>
</tr>
<tr>
<td>D</td>
<td>D</td>
<td>matrix — {'1'}</td>
</tr>
<tr>
<td>InitialCondition</td>
<td>Initial conditions</td>
<td>vector — {'0'}</td>
</tr>
<tr>
<td>SampleTime</td>
<td>Sample time (-1 for inherited)</td>
<td>{'1'}</td>
</tr>
<tr>
<td>StateName</td>
<td>State name</td>
<td>{'''}</td>
</tr>
<tr>
<td>StateMustResolveToSignalObject</td>
<td>State name must resolve to Simulink signal object</td>
<td>{'off'}</td>
</tr>
<tr>
<td>StateSignalObject</td>
<td>Signal object class</td>
<td>Simulink.Signal object</td>
</tr>
<tr>
<td></td>
<td>Code generation storage class</td>
<td>Object of a class that is derived from Simulink.Signal</td>
</tr>
<tr>
<td>StateStorageClass</td>
<td>Code generation storage class</td>
<td>{'Auto'}</td>
</tr>
<tr>
<td>RTWStateStorageTypeQualifier</td>
<td>Code generation storage type qualifier</td>
<td>{'''}</td>
</tr>
<tr>
<td>Discrete Transfer Fcn (DiscreteTransferFcn)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Numerator</td>
<td>Numerator coefficients</td>
<td>vector — {'[1]'}</td>
</tr>
<tr>
<td>Denominator</td>
<td>Denominator coefficients</td>
<td>vector — {'[1 0.5]'}</td>
</tr>
<tr>
<td>InitialStates</td>
<td>Initial states</td>
<td>{'0'}</td>
</tr>
<tr>
<td>SampleTime</td>
<td>Sample time (-1 for inherited)</td>
<td>{'1'}</td>
</tr>
<tr>
<td>Block (Type)/Parameter</td>
<td>Dialog Box Prompt</td>
<td>Values</td>
</tr>
<tr>
<td>------------------------</td>
<td>-----------------------------------------------------------------------------------</td>
<td>---------------------------------------------</td>
</tr>
<tr>
<td>a0EqualsOne</td>
<td>Optimize by skipping divide by leading denominator coefficient (a0)</td>
<td>{'off'}</td>
</tr>
<tr>
<td>NumCoefMin</td>
<td>Numerator coefficient minimum</td>
<td>{'[]}'}</td>
</tr>
<tr>
<td>NumCoefMax</td>
<td>Numerator coefficient maximum</td>
<td>{'[]}'}</td>
</tr>
<tr>
<td>DenCoefMin</td>
<td>Denominator coefficient minimum</td>
<td>{'[]}'}</td>
</tr>
<tr>
<td>DenCoefMax</td>
<td>Denominator coefficient maximum</td>
<td>{'[]}'}</td>
</tr>
<tr>
<td>OutMin</td>
<td>Output minimum</td>
<td>{'[]}'}</td>
</tr>
<tr>
<td>OutMax</td>
<td>Output maximum</td>
<td>{'[]}'}</td>
</tr>
<tr>
<td>StateDataTypeStr</td>
<td>State data type</td>
<td>{'Inherit: Same as input'}</td>
</tr>
<tr>
<td>NumCoefDataTypeStr</td>
<td>Numerator coefficient data type</td>
<td>{'Inherit: Inherit via internal rule'}</td>
</tr>
<tr>
<td>DenCoefDataTypeStr</td>
<td>Denominator coefficient data type</td>
<td>{'Inherit: Inherit via internal rule'}</td>
</tr>
<tr>
<td>NumProductDataTypeStr</td>
<td>Numerator product output data type</td>
<td>{'Inherit: Inherit via internal rule'}</td>
</tr>
<tr>
<td>Block (Type)/Parameter</td>
<td>Dialog Box Prompt</td>
<td>Values</td>
</tr>
<tr>
<td>------------------------------</td>
<td>------------------------------------------</td>
<td>-----------------------------------------------------------------------</td>
</tr>
<tr>
<td>DenProductDataTypeStr</td>
<td>Denominator product output data type</td>
<td>{'Inherit: Inherit via internal rule'}</td>
</tr>
<tr>
<td>NumAccumDataTypeStr</td>
<td>Numerator accumulator data type</td>
<td>{'Inherit: Inherit via internal rule'}</td>
</tr>
<tr>
<td>DenAccumDataTypeStr</td>
<td>Denominator accumulator data type</td>
<td>{'Inherit: Inherit via internal rule'}</td>
</tr>
<tr>
<td>OutDataTypeStr</td>
<td>Output data type</td>
<td>{'Inherit: Inherit via internal rule'}</td>
</tr>
<tr>
<td>LockScale</td>
<td>Lock data type settings against changes by the fixed-point tools</td>
<td>{'off'}</td>
</tr>
<tr>
<td>RndMeth</td>
<td>Integer rounding mode</td>
<td>'Ceiling'</td>
</tr>
<tr>
<td>SaturateOnIntegerOverflow</td>
<td>Saturate on integer overflow</td>
<td>{'off'}</td>
</tr>
<tr>
<td>StateName</td>
<td>State name</td>
<td>{''}</td>
</tr>
<tr>
<td>Block (Type)/Parameter</td>
<td>Dialog Box Prompt</td>
<td>Values</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>--------------------------------------------------------</td>
<td>------------------------------------------------------------------------</td>
</tr>
<tr>
<td>StateMustResolveToSignalObject</td>
<td>State name must resolve to Simulink signal object</td>
<td>{'off'}</td>
</tr>
<tr>
<td>StateSignalObject</td>
<td>Signal object class</td>
<td>Simulink.Signal object</td>
</tr>
<tr>
<td></td>
<td>Code generation storage class</td>
<td>Object of a class that is derived from Simulink.Signal</td>
</tr>
<tr>
<td>StateStorageClass</td>
<td>Code generation storage class</td>
<td>{'Auto'}</td>
</tr>
<tr>
<td>RTWStateStorageTypeQualifier</td>
<td>Code generation storage type qualifier</td>
<td>{''}</td>
</tr>
</tbody>
</table>

**Discrete Zero-Pole (DiscreteZeroPole)**

<table>
<thead>
<tr>
<th>Block (Type)/Parameter</th>
<th>Dialog Box Prompt</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zeros</td>
<td>Zeros</td>
<td>vector — {'[1]' }</td>
</tr>
<tr>
<td>Poles</td>
<td>Poles</td>
<td>vector — {'[0 0.5]'}</td>
</tr>
<tr>
<td>Gain</td>
<td>Gain</td>
<td>{'1'}</td>
</tr>
<tr>
<td>SampleTime</td>
<td>Sample time (-1 for inherited)</td>
<td>{'1'}</td>
</tr>
<tr>
<td>StateName</td>
<td>State name</td>
<td>{''}</td>
</tr>
<tr>
<td>StateMustResolveToSignalObject</td>
<td>State name must resolve to Simulink signal object</td>
<td>{'off'}</td>
</tr>
<tr>
<td>StateSignalObject</td>
<td>Signal object class</td>
<td>Simulink.Signal object</td>
</tr>
<tr>
<td></td>
<td>Code generation storage class</td>
<td>Object of a class that is derived from Simulink.Signal</td>
</tr>
<tr>
<td>StateStorageClass</td>
<td>Code generation storage class</td>
<td>{'Auto'}</td>
</tr>
<tr>
<td>Block (Type)/Parameter</td>
<td>Dialog Box Prompt</td>
<td>Values</td>
</tr>
<tr>
<td>----------------------------------------</td>
<td>----------------------------------------</td>
<td>------------------------------------------------------------------------</td>
</tr>
<tr>
<td>RTWStateStorageType Qualifier</td>
<td>Code generation storage type qualifier</td>
<td>{''}</td>
</tr>
<tr>
<td>Discrete-Time Integrator (DiscreteIntegrator)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IntegratorMethod</td>
<td>Integrator method</td>
<td>{'Integration: Forward Euler'}</td>
</tr>
<tr>
<td>gainval</td>
<td>Gain value</td>
<td>{'1.0'}</td>
</tr>
<tr>
<td>ExternalReset</td>
<td>External reset</td>
<td>{'none'}</td>
</tr>
<tr>
<td>InitialConditionSource</td>
<td>Initial condition source</td>
<td>{'internal'}</td>
</tr>
<tr>
<td>InitialCondition</td>
<td>Initial condition</td>
<td>scalar or vector — {'0'}</td>
</tr>
<tr>
<td>InitialConditionSetting</td>
<td>Initial condition setting</td>
<td>{'State (most efficient)'}</td>
</tr>
<tr>
<td>SampleTime</td>
<td>Sample time (-1 for inherited)</td>
<td>{'1'}</td>
</tr>
<tr>
<td>LimitOutput</td>
<td>Limit output</td>
<td>{'off'}</td>
</tr>
<tr>
<td>UpperSaturationLimit</td>
<td>Upper saturation limit</td>
<td>scalar or vector — {'inf'}</td>
</tr>
<tr>
<td>LowerSaturationLimit</td>
<td>Lower saturation limit</td>
<td>scalar or vector — {'-inf'}</td>
</tr>
<tr>
<td>ShowSaturationPort</td>
<td>Show saturation port</td>
<td>{'off'}</td>
</tr>
<tr>
<td>ShowStatePort</td>
<td>Show state port</td>
<td>{'off'}</td>
</tr>
<tr>
<td>IgnoreLimit</td>
<td>Ignore limit and reset when linearizing</td>
<td>{'off'}</td>
</tr>
<tr>
<td>Block (Type)/Parameter</td>
<td>Dialog Box Prompt</td>
<td>Values</td>
</tr>
<tr>
<td>----------------------------------</td>
<td>----------------------------------------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>OutMin</td>
<td>Output minimum</td>
<td>{'[]'}</td>
</tr>
<tr>
<td>OutMax</td>
<td>Output maximum</td>
<td>{'[]'}</td>
</tr>
<tr>
<td>OutDataTypeStr</td>
<td>Output data type</td>
<td>{'Inherit: Inherit via internal rule'}</td>
</tr>
<tr>
<td>LockScale</td>
<td>Lock output data type setting against changes by the fixed-point tools</td>
<td>{'off'}</td>
</tr>
<tr>
<td>RndMeth</td>
<td>Integer rounding mode</td>
<td>'Ceiling'</td>
</tr>
<tr>
<td>SaturateOnInteger Overflow</td>
<td>Saturate on integer overflow</td>
<td>{'off'}</td>
</tr>
<tr>
<td>StateName</td>
<td>State name</td>
<td>{''}</td>
</tr>
<tr>
<td>StateMustResolveToSignalObject</td>
<td>State name must resolve to Simulink signal object</td>
<td>{'off'}</td>
</tr>
<tr>
<td>StateSignalObject</td>
<td>Signal object class</td>
<td>Simulink.Signal object</td>
</tr>
<tr>
<td>StateStorageClass</td>
<td>Code generation storage class</td>
<td>Object of a class that is derived from Simulink.Signal</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{'Auto'}</td>
</tr>
</tbody>
</table>
### RTWStateStorageType Qualifier

**Dialog Box Prompt**: Code generation storage type qualifier

**Values**: 

- `{''}`

---

### First-Order Hold (First-Order Hold) (masked subsystem)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Dialog Box Prompt</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Ts</strong></td>
<td>Sample time</td>
<td>{'1'}</td>
</tr>
</tbody>
</table>

---

### Memory (Memory)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Dialog Box Prompt</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>InitialCondition</strong></td>
<td>Initial condition</td>
<td>scalar or vector — {'0'}</td>
</tr>
<tr>
<td><strong>InheritSampleTime</strong></td>
<td>Inherit sample time</td>
<td>{'off'}</td>
</tr>
<tr>
<td><strong>LinearizeMemory</strong></td>
<td>Direct feedthrough of input during linearization</td>
<td>{'off'}</td>
</tr>
<tr>
<td><strong>LinearizeAsDelay</strong></td>
<td>Treat as a unit delay when linearizing with discrete sample time</td>
<td>{'off'}</td>
</tr>
<tr>
<td><strong>StateName</strong></td>
<td>State name</td>
<td>{'''}</td>
</tr>
<tr>
<td><strong>StateMustResolveTo SignalObject</strong></td>
<td>State name must resolve to Simulink signal object</td>
<td>{'off'}</td>
</tr>
<tr>
<td><strong>StateSignalObject</strong></td>
<td>Signal object class</td>
<td>Simulink.Signal object</td>
</tr>
<tr>
<td></td>
<td>Code generation storage class</td>
<td>Object of a class that is derived from Simulink.Signal</td>
</tr>
<tr>
<td><strong>StateStorageClass</strong></td>
<td>Code generation storage class</td>
<td>{'Auto'}</td>
</tr>
</tbody>
</table>

---

### Tapped Delay (S-Function) (Tapped Delay Line) (masked subsystem)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Dialog Box Prompt</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>vinit</strong></td>
<td>Initial condition</td>
<td>{'0.0'}</td>
</tr>
<tr>
<td><strong>samptime</strong></td>
<td>Sample time</td>
<td>{'-1'}</td>
</tr>
<tr>
<td><strong>NumDelays</strong></td>
<td>Number of delays</td>
<td>{'4'}</td>
</tr>
<tr>
<td>Block (Type)/Parameter</td>
<td>Dialog Box Prompt</td>
<td>Values</td>
</tr>
<tr>
<td>-----------------------</td>
<td>-------------------------------------------------------</td>
<td>-------------------------</td>
</tr>
<tr>
<td>DelayOrder</td>
<td>Order output vector starting with</td>
<td>{'Oldest'}</td>
</tr>
<tr>
<td>includeCurrent</td>
<td>Include current input in output vector</td>
<td>{'off'}</td>
</tr>
</tbody>
</table>

Transfer Fcn First Order (First Order Transfer Fcn) (masked subsystem)

<table>
<thead>
<tr>
<th>PoleZ</th>
<th>Pole (in Z plane)</th>
<th>{'0.95'}</th>
</tr>
</thead>
<tbody>
<tr>
<td>ICPrevOutput</td>
<td>Initial condition for previous output</td>
<td>{'0.0'}</td>
</tr>
<tr>
<td>RndMeth</td>
<td>Integer rounding mode</td>
<td>'Ceiling'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{'Floor'}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>'Round'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>'Zero'</td>
</tr>
<tr>
<td>DoSatur</td>
<td>Saturate to max or min when overflows occur</td>
<td>{'off'}</td>
</tr>
</tbody>
</table>

Transfer Fcn Lead or Lag (Lead or Lag Compensator) (masked subsystem)

<table>
<thead>
<tr>
<th>PoleZ</th>
<th>Pole of compensator (in Z plane)</th>
<th>{'0.95'}</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZeroZ</td>
<td>Zero of compensator (in Z plane)</td>
<td>{'0.75'}</td>
</tr>
<tr>
<td>ICPrevOutput</td>
<td>Initial condition for previous output</td>
<td>{'0.0'}</td>
</tr>
<tr>
<td>ICPrevInput</td>
<td>Initial condition for previous input</td>
<td>{'0.0'}</td>
</tr>
<tr>
<td>RndMeth</td>
<td>Integer rounding mode</td>
<td>'Ceiling'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{'Floor'}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>'Round'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>'Zero'</td>
</tr>
<tr>
<td>DoSatur</td>
<td>Saturate to max or min when overflows occur</td>
<td>{'off'}</td>
</tr>
</tbody>
</table>

Transfer Fcn Real Zero (Transfer Fcn Real Zero) (masked subsystem)

<table>
<thead>
<tr>
<th>ZeroZ</th>
<th>Zero (in Z plane)</th>
<th>{'0.75'}</th>
</tr>
</thead>
<tbody>
<tr>
<td>ICPrevInput</td>
<td>Initial condition for previous input</td>
<td>{'0.0'}</td>
</tr>
<tr>
<td>Block (Type)/Parameter</td>
<td>Dialog Box Prompt</td>
<td>Values</td>
</tr>
<tr>
<td>------------------------</td>
<td>-------------------</td>
<td>--------</td>
</tr>
<tr>
<td>RndMeth</td>
<td>Integer rounding mode</td>
<td>'Ceiling'</td>
</tr>
<tr>
<td>DoSatur</td>
<td>Saturate to max or min when overflows occur</td>
<td>{'off'}</td>
</tr>
<tr>
<td>Unit Delay (UnitDelay)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>InitialCondition</td>
<td>Initial condition</td>
<td>scalar or vector — {'0'}</td>
</tr>
<tr>
<td>InputProcessing</td>
<td>Input processing</td>
<td>'Columns as channels (frame based)'</td>
</tr>
<tr>
<td>SampleTime</td>
<td>Sample time (-1 for inherited)</td>
<td>{'-1'}</td>
</tr>
<tr>
<td>StateName</td>
<td>State name</td>
<td>{''}</td>
</tr>
<tr>
<td>StateMustResolveToSignalObject</td>
<td>State name must resolve to Simulink signal object</td>
<td>{'off'}</td>
</tr>
<tr>
<td>StateSignalObject</td>
<td>Signal object class</td>
<td>Simulink.Signal object</td>
</tr>
<tr>
<td>StateStorageClass</td>
<td>Code generation storage class</td>
<td>Object of a class that is derived from Simulink.Signal</td>
</tr>
<tr>
<td>CodeGenStateStorageTypeQualifier</td>
<td>Code generation storage type qualifier</td>
<td>{''}</td>
</tr>
<tr>
<td>Zero-Order Hold (ZeroOrderHold)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SampleTime</td>
<td>Sample time (-1 for inherited)</td>
<td>{'1'}</td>
</tr>
</tbody>
</table>
## Logic and Bit Operations Library Block Parameters

<table>
<thead>
<tr>
<th>Block (Type)/Parameter</th>
<th>Dialog Box Prompt</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Bit Clear (Bit Clear)</strong> (masked subsystem)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>iBit</td>
<td>Index of bit (0 is least significant)</td>
<td>{'0'}</td>
</tr>
<tr>
<td><strong>Bit Set (Bit Set)</strong> (masked subsystem)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>iBit</td>
<td>Index of bit (0 is least significant)</td>
<td>{'0'}</td>
</tr>
<tr>
<td><strong>Bitwise Operator (S-Function)</strong> (Bitwise Operator) (masked subsystem)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>logicop</td>
<td>Operator</td>
<td>{'AND'}</td>
</tr>
<tr>
<td>UseBitMask</td>
<td>Use bit mask ...</td>
<td>'off'</td>
</tr>
<tr>
<td>NumInputPorts</td>
<td>Number of input ports</td>
<td>{'1'}</td>
</tr>
<tr>
<td>BitMask</td>
<td>Bit Mask</td>
<td>{'bin2dec('11011001')'}</td>
</tr>
<tr>
<td>BitMaskRealWorld</td>
<td>Treat mask as</td>
<td>'Real World Value'</td>
</tr>
<tr>
<td><strong>Combinatorial Logic (CombinatorialLogic)</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TruthTable</td>
<td>Truth table</td>
<td>{'[0 0;0 1;0 1;0 1;0 1;1 0;1 0;1 1]'}</td>
</tr>
<tr>
<td>SampleTime</td>
<td>Sample time (-1 for inherited)</td>
<td>{'-1'}</td>
</tr>
<tr>
<td><strong>Compare To Constant (Compare To Constant)</strong> (masked subsystem)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>relop</td>
<td>Operator</td>
<td>'=='</td>
</tr>
<tr>
<td>const</td>
<td>Constant value</td>
<td>{'3.0'}</td>
</tr>
<tr>
<td>OutDataTypeStr</td>
<td>Output data type</td>
<td>{'boolean'}</td>
</tr>
<tr>
<td>ZeroCross</td>
<td>Enable zero-crossing detection</td>
<td>'off'</td>
</tr>
<tr>
<td><strong>Compare To Zero (Compare To Zero)</strong> (masked subsystem)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>relop</td>
<td>Operator</td>
<td>'=='</td>
</tr>
<tr>
<td>OutDataTypeStr</td>
<td>Output data type</td>
<td>{'boolean'}</td>
</tr>
<tr>
<td>ZeroCross</td>
<td>Enable zero-crossing detection</td>
<td>'off'</td>
</tr>
<tr>
<td><strong>Block (Type)/Parameter</strong></td>
<td><strong>Dialog Box Prompt</strong></td>
<td><strong>Values</strong></td>
</tr>
<tr>
<td>---------------------------</td>
<td>----------------------</td>
<td>-----------</td>
</tr>
<tr>
<td>Detect Change (Detect Change) (masked subsystem)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>vinit</td>
<td>Initial condition</td>
<td>{'0'}</td>
</tr>
<tr>
<td>OutDataTypeStr</td>
<td>Output data type</td>
<td>{'boolean'}</td>
</tr>
<tr>
<td>Detect Decrease (Detect Decrease) (masked subsystem)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>vinit</td>
<td>Initial condition</td>
<td>{'0.0'}</td>
</tr>
<tr>
<td>OutDataTypeStr</td>
<td>Output data type</td>
<td>{'boolean'}</td>
</tr>
<tr>
<td>Detect Fall Negative (Detect Fall Negative) (masked subsystem)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>vinit</td>
<td>Initial condition</td>
<td>{'0'}</td>
</tr>
<tr>
<td>OutDataTypeStr</td>
<td>Output data type</td>
<td>{'boolean'}</td>
</tr>
<tr>
<td>Detect Fall Nonpositive (Detect Fall Nonpositive) (masked subsystem)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>vinit</td>
<td>Initial condition</td>
<td>{'0'}</td>
</tr>
<tr>
<td>OutDataTypeStr</td>
<td>Output data type</td>
<td>{'boolean'}</td>
</tr>
<tr>
<td>Detect Increase (Detect Increase) (masked subsystem)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>vinit</td>
<td>Initial condition</td>
<td>{'0.0'}</td>
</tr>
<tr>
<td>OutDataTypeStr</td>
<td>Output data type</td>
<td>{'boolean'}</td>
</tr>
<tr>
<td>Detect Rise Nonnegative (Detect Rise Nonnegative) (masked subsystem)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>vinit</td>
<td>Initial condition</td>
<td>{'0'}</td>
</tr>
<tr>
<td>OutDataTypeStr</td>
<td>Output data type</td>
<td>{'boolean'}</td>
</tr>
<tr>
<td>Detect Rise Positive (Detect Rise Positive) (masked subsystem)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>vinit</td>
<td>Initial condition</td>
<td>{'0'}</td>
</tr>
<tr>
<td>OutDataTypeStr</td>
<td>Output data type</td>
<td>{'boolean'}</td>
</tr>
<tr>
<td>Extract Bits (Extract Bits) (masked subsystem)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bitsToExtract</td>
<td>Bits to extract</td>
<td>{'Upper half'}</td>
</tr>
<tr>
<td>numBits</td>
<td>Number of bits</td>
<td>{'8'}</td>
</tr>
<tr>
<td>Block (Type)/Parameter</td>
<td>Dialog Box Prompt</td>
<td>Values</td>
</tr>
<tr>
<td>---------------------------</td>
<td>------------------------------------------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>bitIdxRange</td>
<td>Bit indices ([start end], 0-based relative to LSB)</td>
<td>{'[0 7]'}</td>
</tr>
<tr>
<td>outScalingMode</td>
<td>Output scaling mode</td>
<td>{'Preserve fixed-point scaling'}</td>
</tr>
<tr>
<td>Interval Test (Interval Test) (masked subsystem)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IntervalClosedRight</td>
<td>Interval closed on right</td>
<td>'off'</td>
</tr>
<tr>
<td>uplimit</td>
<td>Upper limit</td>
<td>{'0.5'}</td>
</tr>
<tr>
<td>IntervalClosedLeft</td>
<td>Interval closed on left</td>
<td>'off'</td>
</tr>
<tr>
<td>lowlimit</td>
<td>Lower limit</td>
<td>{'-0.5'}</td>
</tr>
<tr>
<td>OutDataTypeStr</td>
<td>Output data type</td>
<td>{'boolean'}</td>
</tr>
<tr>
<td>Interval Test Dynamic (Interval Test Dynamic) (masked subsystem)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IntervalClosedRight</td>
<td>Interval closed on right</td>
<td>'off'</td>
</tr>
<tr>
<td>IntervalClosedLeft</td>
<td>Interval closed on left</td>
<td>'off'</td>
</tr>
<tr>
<td>OutDataTypeStr</td>
<td>Output data type</td>
<td>{'boolean'}</td>
</tr>
<tr>
<td>Logical Operator (Logic)</td>
<td>Operator</td>
<td>{'AND'}</td>
</tr>
<tr>
<td>Inputs</td>
<td>Number of input ports</td>
<td>{'2'}</td>
</tr>
<tr>
<td>IconShape</td>
<td>Icon shape</td>
<td>{'rectangular'}</td>
</tr>
<tr>
<td>SampleTime</td>
<td>Sample time (-1 for inherited)</td>
<td>{'-1'}</td>
</tr>
<tr>
<td>AllPortsSameDT</td>
<td>Require all inputs and output to have the same data type</td>
<td>{'off'}</td>
</tr>
<tr>
<td>OutDataTypeStr</td>
<td>Output data type</td>
<td>'Inherit: Logical (see Configuration Parameters: Optimization)'</td>
</tr>
<tr>
<td>Block (Type)/Parameter</td>
<td>Dialog Box Prompt</td>
<td>Values</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>----------------------------------</td>
<td>---------------------------------------------</td>
</tr>
<tr>
<td>Relational Operator (RelationalOperator)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Operator</strong></td>
<td>Relational operator</td>
<td>`'=='</td>
</tr>
<tr>
<td><strong>ZeroCross</strong></td>
<td>Enable zero-crossing detection</td>
<td>'off'</td>
</tr>
<tr>
<td><strong>SampleTime</strong></td>
<td>Sample time (-1 for inherited)</td>
<td>{'-1'}</td>
</tr>
<tr>
<td><strong>InputSameDT</strong></td>
<td>Require all inputs to have the same data type</td>
<td>{'off'}</td>
</tr>
<tr>
<td><strong>OutDataTypeStr</strong></td>
<td>Output data type</td>
<td>'Inherit: Logical (see Configuration Parameters: Optimization)'</td>
</tr>
<tr>
<td>Shift Arithmetic (ArithShift)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>BitShiftNumberSource</strong></td>
<td>Bits to shift &gt; Source</td>
<td>{'Dialog'}</td>
</tr>
<tr>
<td><strong>BitShiftDirection</strong></td>
<td>Bits to shift &gt; Direction</td>
<td>'Left'</td>
</tr>
<tr>
<td><strong>BitShiftNumber</strong></td>
<td>Bits to shift &gt; Number</td>
<td>{'8'}</td>
</tr>
<tr>
<td><strong>BinPtShiftNumber</strong></td>
<td>Binary points to shift &gt; Number</td>
<td>{'0'}</td>
</tr>
<tr>
<td><strong>DiagnosticForOORShift</strong></td>
<td>Diagnostic for out-of-range shift value</td>
<td>{'None'}</td>
</tr>
<tr>
<td><strong>CheckOORBitShift</strong></td>
<td>Check for out-of-range 'Bits to shift' in generated code</td>
<td>{'off'}</td>
</tr>
<tr>
<td><strong>nBitShiftRight</strong></td>
<td>Deprecated in R2011a</td>
<td></td>
</tr>
<tr>
<td><strong>nBinPtShiftRight</strong></td>
<td>Deprecated in R2011a</td>
<td></td>
</tr>
</tbody>
</table>
## Lookup Table Library Block Parameters

<table>
<thead>
<tr>
<th>Block (Type)/Parameter</th>
<th>Dialog Box Prompt</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Cosine (Cosine) (masked subsystem)</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Formula</td>
<td>Output formula</td>
<td>'sin(2<em>pi</em>u)'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{'cos(2<em>pi</em>u)'}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>'exp(j<em>2</em>pi*u)'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>'sin(2<em>pi</em>u) and cos(2<em>pi</em>u)'</td>
</tr>
<tr>
<td>NumDataPoints</td>
<td>Number of data points for lookup table</td>
<td>{'(2^5)+1'}</td>
</tr>
<tr>
<td>OutputWordLength</td>
<td>Output word length</td>
<td>{'16'}</td>
</tr>
<tr>
<td>InternalRulePriority</td>
<td>Internal rule priority for lookup table</td>
<td>{'Speed'}</td>
</tr>
<tr>
<td><strong>Direct Lookup Table (n-D) (LookupNDDirect)</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NumberOfTableDimensions</td>
<td>Number of table dimensions</td>
<td>'1'</td>
</tr>
<tr>
<td>InputsSelectThisObjectFromTable</td>
<td>Inputs select this object from table</td>
<td>{'Element'}</td>
</tr>
<tr>
<td>TableIsInput</td>
<td>Make table an input</td>
<td>{'off'}</td>
</tr>
<tr>
<td>Table</td>
<td>Table data</td>
<td>{'[4 5 6;16 19 20;10 18 23]'}</td>
</tr>
<tr>
<td>DiagnosticForOutOfRangeInput</td>
<td>Diagnostic for out-of-range input</td>
<td>'None'</td>
</tr>
<tr>
<td>SampleTime</td>
<td>Sample time (-1 for inherited)</td>
<td>{'-1'}</td>
</tr>
<tr>
<td>TableMin</td>
<td>Table minimum</td>
<td>{'[]'}</td>
</tr>
<tr>
<td>TableMax</td>
<td>Table maximum</td>
<td>{'[]'}</td>
</tr>
<tr>
<td>TableDataTypeStr</td>
<td>Table data type</td>
<td>{'Inherit: Inherit from Table data'}</td>
</tr>
<tr>
<td>Block (Type)/Parameter</td>
<td>Dialog Box Prompt</td>
<td>Values</td>
</tr>
<tr>
<td>------------------------</td>
<td>-------------------</td>
<td>--------</td>
</tr>
<tr>
<td>LockScale</td>
<td>Lock data type settings against changes by the fixed-point tools</td>
<td>{'off'}</td>
</tr>
<tr>
<td>maskTabDims</td>
<td>Deprecated in R2009b</td>
<td></td>
</tr>
<tr>
<td>explicitNumDims</td>
<td>Deprecated in R2009b</td>
<td></td>
</tr>
<tr>
<td>outDims</td>
<td>Deprecated in R2009b</td>
<td></td>
</tr>
<tr>
<td>tabIsInput</td>
<td>Deprecated in R2009b</td>
<td></td>
</tr>
<tr>
<td>mxTable</td>
<td>Deprecated in R2009b</td>
<td></td>
</tr>
<tr>
<td>clipFlag</td>
<td>Deprecated in R2009b</td>
<td></td>
</tr>
<tr>
<td>samptime</td>
<td>Deprecated in R2009b</td>
<td></td>
</tr>
<tr>
<td>Interpolation Using Prelookup (Interpolation_n-D)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NumberOfTableDimensions</td>
<td>Number of table dimensions</td>
<td>'1'</td>
</tr>
<tr>
<td>Table</td>
<td>Table data &gt; Value</td>
<td>{'sqrt([1:11]' * [1:11])'}</td>
</tr>
<tr>
<td>TableSource</td>
<td>Table data &gt; Source</td>
<td>{'Dialog'}</td>
</tr>
<tr>
<td>TableSpecification</td>
<td>Specification</td>
<td>{'Explicit values'}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>To set this parameter from 'Explicit values' to 'Lookup table object', use the same call to set_param to set the parameter LookupTableObject. For example:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>set_param('myModel/myInterpBlock',...</td>
</tr>
<tr>
<td></td>
<td></td>
<td>'TableSpecification',...</td>
</tr>
<tr>
<td></td>
<td></td>
<td>'Lookup table object',...</td>
</tr>
<tr>
<td></td>
<td></td>
<td>'LookupTableObject','myLUTOobject')</td>
</tr>
<tr>
<td>LookupTableObject</td>
<td>Name of lookup table object</td>
<td>{''}</td>
</tr>
<tr>
<td>InterpMethod</td>
<td>Interpolation method</td>
<td>'Flat'</td>
</tr>
<tr>
<td>ExtrapMethod</td>
<td>Extrapolation method</td>
<td>'Clip'</td>
</tr>
<tr>
<td>Block (Type)/Parameter</td>
<td>Dialog Box Prompt</td>
<td>Values</td>
</tr>
<tr>
<td>------------------------</td>
<td>-------------------</td>
<td>--------</td>
</tr>
<tr>
<td>ValidIndexMayReachLast</td>
<td>Valid index input may reach last index</td>
<td>{'off'}</td>
</tr>
<tr>
<td>DiagnosticForOutOfRangeInput</td>
<td>Diagnostic for out-of-range input</td>
<td>{'None'}</td>
</tr>
<tr>
<td>RemoveProtectionIndex</td>
<td>Remove protection against out-of-range index in generated code</td>
<td>{'off'}</td>
</tr>
<tr>
<td>NumSelectionDims</td>
<td>Number of sub-table selection dimensions</td>
<td>{'0'}</td>
</tr>
<tr>
<td>SampleTime</td>
<td>Sample time (-1 for inherited)</td>
<td>{'-1'}</td>
</tr>
<tr>
<td>TableDataTypeStr</td>
<td>Table data &gt; Data Type</td>
<td>'Inherit: Inherit from 'Table data''</td>
</tr>
<tr>
<td>TableMin</td>
<td>Table data &gt; Minimum</td>
<td>{'[]'}</td>
</tr>
<tr>
<td>TableMax</td>
<td>Table data &gt; Maximum</td>
<td>{'[]'}</td>
</tr>
<tr>
<td>IntermediateResultsDataTypeStr</td>
<td>Intermediate results &gt; Data Type</td>
<td>{'Inherit: Inherit via internal rule'}</td>
</tr>
<tr>
<td>Block (Type)/Parameter</td>
<td>Dialog Box Prompt</td>
<td>Values</td>
</tr>
<tr>
<td>------------------------</td>
<td>------------------</td>
<td>--------</td>
</tr>
<tr>
<td>OutDataTypeStr</td>
<td>Output &gt; Data Type</td>
<td>'Inherit: Inherit via back propagation'</td>
</tr>
<tr>
<td>OutMin</td>
<td>Output &gt; Minimum</td>
<td>{'[]'}</td>
</tr>
<tr>
<td>OutMax</td>
<td>Output &gt; Maximum</td>
<td>{'[]'}</td>
</tr>
<tr>
<td>InternalRulePriority</td>
<td>Internal rule priority</td>
<td>{'Speed'}</td>
</tr>
<tr>
<td>LockScale</td>
<td>Lock data type settings against changes by the fixed-point tools</td>
<td>{'off'}</td>
</tr>
<tr>
<td>RndMeth</td>
<td>Integer rounding mode</td>
<td>'Ceiling'</td>
</tr>
<tr>
<td>SaturateOnIntegerOverflow</td>
<td>Saturate on integer overflow</td>
<td>{'off'}</td>
</tr>
<tr>
<td>CheckIndexInCode</td>
<td>Deprecated in R2011a</td>
<td></td>
</tr>
</tbody>
</table>

n-D Lookup Table, 1-D Lookup Table, 2-D Lookup Table (Lookup_n-D)

<p>| NumberOfTableDimensions | Number of table dimensions | '1' | '2' | '3' | '4'. Default is ‘1’ for 1-D Lookup Table, ‘2’ for 2-D Lookup Table, ‘3’ for n-D Lookup Table. |</p>
<table>
<thead>
<tr>
<th>Block (Type)/Parameter</th>
<th>Dialog Box Prompt</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>DataSpecification</td>
<td>(n-D Lookup Table) Data specification</td>
<td>{'Table and breakpoints'}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>To set this parameter from 'Table and breakpoints' to 'Lookup table object', use the same call to set_param to set the parameter LookupTableObject. For example: set_param('myModel/myLookupBlock',... 'DataSpecification','Lookup table object',... 'LookupTableObject','myLUTObject')</td>
</tr>
<tr>
<td>LookupTableObject</td>
<td>Name of lookup table object.</td>
<td>{''}</td>
</tr>
<tr>
<td>Table</td>
<td>Table data</td>
<td>{'reshape(repmat([4 5 6;16 19 20;10 18 23],1,2),[3,3,2])'}</td>
</tr>
<tr>
<td>BreakpointsSpecification</td>
<td>Breakpoints specification</td>
<td>{'Explicit values'}</td>
</tr>
<tr>
<td>BreakpointsForDimension1FirstPoint</td>
<td>First point</td>
<td>{'1'}</td>
</tr>
<tr>
<td>BreakpointsForDimension2FirstPoint</td>
<td>First point</td>
<td>{'1'}</td>
</tr>
<tr>
<td>BreakpointsForDimension3FirstPoint</td>
<td>First point</td>
<td>{'1'}</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>BreakpointsForDimension30FirstPoint</td>
<td>First point</td>
<td>{'1'}</td>
</tr>
<tr>
<td>BreakpointsForDimension1Spacing</td>
<td>Spacing</td>
<td>{'1'}</td>
</tr>
<tr>
<td>BreakpointsForDimension2Spacing</td>
<td>Spacing</td>
<td>{'1'}</td>
</tr>
<tr>
<td>BreakpointsForDimension3Spacing</td>
<td>Spacing</td>
<td>{'1'}</td>
</tr>
<tr>
<td>Block (Type)/Parameter</td>
<td>Dialog Box Prompt</td>
<td>Values</td>
</tr>
<tr>
<td>------------------------</td>
<td>-------------------</td>
<td>--------</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>BreakpointsFor-Dimension30Spacing</td>
<td>Spacing</td>
<td>{'1'}</td>
</tr>
<tr>
<td>BreakpointsFor-Dimension1</td>
<td>Breakpoints 1</td>
<td>{'[10,22,31]'}</td>
</tr>
<tr>
<td>BreakpointsForDimension2</td>
<td>Breakpoints 2</td>
<td>{'[10,22,31]'}</td>
</tr>
<tr>
<td>BreakpointsForDimension3</td>
<td>Breakpoints 3</td>
<td>{'[5, 7]'}</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>BreakpointsFor-Dimension30</td>
<td>Breakpoints 30</td>
<td>{'[1:3]'}</td>
</tr>
<tr>
<td>SampleTime</td>
<td>Sample time (-1 for inherited)</td>
<td>{'-1'}</td>
</tr>
<tr>
<td>InterpMethod</td>
<td>Interpolation method</td>
<td>'Flat'</td>
</tr>
<tr>
<td>ExtrapMethod</td>
<td>Extrapolation method</td>
<td>'Clip'</td>
</tr>
<tr>
<td>UseLastTableValue</td>
<td>Use last table value for inputs at or above last breakpoint</td>
<td>{'off'}</td>
</tr>
<tr>
<td>DiagnosticForOutOfRangeInput</td>
<td>Diagnostic for out-of-range input</td>
<td>{'None'}</td>
</tr>
<tr>
<td>RemoveProtectionInput</td>
<td>Remove protection against out-of-range input in generated code</td>
<td>{'off'}</td>
</tr>
<tr>
<td>IndexSearchMethod</td>
<td>Index search method</td>
<td>'Evenly spaced points'</td>
</tr>
<tr>
<td>BeginIndexSearchUsingPreviousIndexResult</td>
<td>Begin index search using previous index result</td>
<td>{'off'}</td>
</tr>
<tr>
<td>Block (Type)/Parameter</td>
<td>Dialog Box Prompt</td>
<td>Values</td>
</tr>
<tr>
<td>------------------------</td>
<td>-------------------</td>
<td>--------</td>
</tr>
<tr>
<td>UseOneInputPortForAll InputData</td>
<td>Use one input port for all input data</td>
<td>{'off'}</td>
</tr>
<tr>
<td>SupportTunableTableSize</td>
<td>Support tunable table size in code generation</td>
<td>{'off'}</td>
</tr>
<tr>
<td>MaximumIndicesForEach Dimension</td>
<td>Maximum indices for each dimension</td>
<td>{'[]'}</td>
</tr>
<tr>
<td>TableDataTypeStr</td>
<td>Table data &gt; Data Type</td>
<td>'Inherit: Inherit from 'Table data''</td>
</tr>
<tr>
<td>TableMin</td>
<td>Table data &gt; Minimum</td>
<td>{'[]'}</td>
</tr>
<tr>
<td>TableMax</td>
<td>Table data &gt; Maximum</td>
<td>{'[]'}</td>
</tr>
<tr>
<td>BreakpointsForDimension1 DataTypeStr</td>
<td>Breakpoints 1 &gt; Data Type</td>
<td>{'Inherit: Same as corresponding input'}</td>
</tr>
<tr>
<td>BreakpointsForDimension1 Min</td>
<td>Breakpoints 1 &gt; Minimum</td>
<td>{'[]'}</td>
</tr>
<tr>
<td>BreakpointsForDimension1 Max</td>
<td>Breakpoints 1 &gt; Maximum</td>
<td>{'[]'}</td>
</tr>
<tr>
<td>Block (Type)/Parameter</td>
<td>Dialog Box Prompt</td>
<td>Values</td>
</tr>
<tr>
<td>-----------------------</td>
<td>-------------------</td>
<td>--------</td>
</tr>
<tr>
<td>BreakpointsForDimension2 DataTypeStr</td>
<td>Breakpoints 2 &gt; Data Type</td>
<td>{'Inherit: Same as corresponding input'}</td>
</tr>
<tr>
<td>BreakpointsForDimension2 Min</td>
<td>Breakpoints 2 &gt; Minimum</td>
<td>{'[]'}</td>
</tr>
<tr>
<td>BreakpointsForDimension2 Max</td>
<td>Breakpoints 2 &gt; Maximum</td>
<td>{'[]'}</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>BreakpointsForDimension30 DataTypeStr</td>
<td>Breakpoints 30 &gt; Data Type</td>
<td>{'Inherit: Same as corresponding input'}</td>
</tr>
<tr>
<td>BreakpointsForDimension30 Min</td>
<td>Breakpoints 30 &gt; Minimum</td>
<td>{'[]'}</td>
</tr>
<tr>
<td>BreakpointsForDimension30 Max</td>
<td>Breakpoints 30 &gt; Maximum</td>
<td>{'[]'}</td>
</tr>
<tr>
<td>Block (Type)/Parameter</td>
<td>Dialog Box Prompt</td>
<td>Values</td>
</tr>
<tr>
<td>--------------------------------------</td>
<td>-------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>FractionDataTypeStr</td>
<td>Fraction &gt; Data Type</td>
<td>{'Inherit: Inherit via internal rule'}</td>
</tr>
<tr>
<td>IntermediateResults DataTypeStr</td>
<td>Intermediate results &gt; Data Type</td>
<td>'Inherit: Inherit via internal rule'</td>
</tr>
<tr>
<td>OutDataTypeStr</td>
<td>Output &gt; Data Type</td>
<td>'Inherit: Inherit via back propagation'</td>
</tr>
<tr>
<td>OutMin</td>
<td>Output &gt; Minimum</td>
<td>{'[']'}</td>
</tr>
<tr>
<td>OutMax</td>
<td>Output &gt; Maximum</td>
<td>{'[']'}</td>
</tr>
<tr>
<td>InternalRulePriority</td>
<td>Internal rule priority</td>
<td>{'Speed'}</td>
</tr>
<tr>
<td>InputSameDT</td>
<td>Require all inputs to have the same data type</td>
<td>'off'</td>
</tr>
<tr>
<td>LockScale</td>
<td>Lock data type settings against changes by the fixed-point tools</td>
<td>{'off'}</td>
</tr>
<tr>
<td>RndMeth</td>
<td>Integer rounding mode</td>
<td>'Ceiling'</td>
</tr>
<tr>
<td>SaturateOnIntegerOverflow</td>
<td>Saturate on integer overflow</td>
<td>{'off'}</td>
</tr>
<tr>
<td>ProcessOutOfRangeInput</td>
<td>Deprecated in R2009b</td>
<td></td>
</tr>
<tr>
<td>Block (Type)/Parameter</td>
<td>Dialog Box Prompt</td>
<td>Values</td>
</tr>
<tr>
<td>------------------------------</td>
<td>----------------------------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>Lookup Table Dynamic (Lookup Table Dynamic) (masked subsystem)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LookUpMeth</td>
<td>Lookup Method</td>
<td>'Interpolation-Extrapolation'</td>
</tr>
<tr>
<td>OutDataTypeStr</td>
<td>Output data type</td>
<td>{'fixdt('double')'}</td>
</tr>
<tr>
<td>OutputDataTypeScaling Mode</td>
<td>Deprecated in R2007b</td>
<td></td>
</tr>
<tr>
<td>OutDataType</td>
<td>Deprecated in R2007b</td>
<td></td>
</tr>
<tr>
<td>OutScaling</td>
<td>Deprecated in R2007b</td>
<td></td>
</tr>
<tr>
<td>LockScale</td>
<td>Lock output data type</td>
<td>{'off'}</td>
</tr>
<tr>
<td></td>
<td>setting against changes by the fixed-point tools</td>
<td></td>
</tr>
<tr>
<td>RndMeth</td>
<td>Integer rounding mode</td>
<td>'Ceiling'</td>
</tr>
<tr>
<td>DoSatur</td>
<td>Saturate to max or min when overflows occur</td>
<td>{'off'}</td>
</tr>
<tr>
<td>Prelookup (PreLookup)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Block (Type)/Parameter</td>
<td>Dialog Box Prompt</td>
<td>Values</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>---------------------------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>BreakpointsSpecification</td>
<td>Specification</td>
<td>{'Explicit values'}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>To set this parameter from 'Explicit values' or 'Even spacing' to 'Breakpoint object', use the same call to <code>set_param</code> to set the parameter BreakpointObject. For example: <code>set_param('myModel/myPrelookupBlock','BreakpointsSpecification','Breakpoint object','BreakpointObject','myBPObject')</code></td>
</tr>
<tr>
<td>BreakpointObject</td>
<td>Name of breakpoint object</td>
<td>{''       }</td>
</tr>
<tr>
<td>BreakpointsFirstPoint</td>
<td>First point</td>
<td>{'10'}</td>
</tr>
<tr>
<td>BreakpointsSpacing</td>
<td>Spacing</td>
<td>{'10'}</td>
</tr>
<tr>
<td>BreakpointsNumPoints</td>
<td>Number of points</td>
<td>{'11'}</td>
</tr>
<tr>
<td>BreakpointsData</td>
<td>Value</td>
<td>{'[10:10:110]'}</td>
</tr>
<tr>
<td>BreakpointsDataSource</td>
<td>Source</td>
<td>{'Dialog'}</td>
</tr>
<tr>
<td>IndexSearchMethod</td>
<td>Index search method</td>
<td>'Evenly spaced points'</td>
</tr>
<tr>
<td>BeginIndexSearchUsingPreviousIndexResult</td>
<td>Begin index search using previous index result</td>
<td>{'off'}</td>
</tr>
<tr>
<td>OutputOnlyTheIndex</td>
<td>Output only the index</td>
<td>{'off'}</td>
</tr>
<tr>
<td>ExtrapMethod</td>
<td>Extrapolation method</td>
<td>'Clip'</td>
</tr>
<tr>
<td>UseLastBreakpoint</td>
<td>Use last breakpoint for input at or above upper limit</td>
<td>{'off'}</td>
</tr>
<tr>
<td>DiagnosticForOutOfRangeInput</td>
<td>Diagnostic for out-of-range input</td>
<td>{'None'}</td>
</tr>
<tr>
<td>Block (Type)/Parameter</td>
<td>Dialog Box Prompt</td>
<td>Values</td>
</tr>
<tr>
<td>-------------------------------------</td>
<td>-----------------------------------------------------------------------------------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>RemoveProtectionInput</td>
<td>Remove protection against out-of-range input in generated code</td>
<td>'off'</td>
</tr>
<tr>
<td>SampleTime</td>
<td>Sample time (-1 for inherited)</td>
<td>'-1'</td>
</tr>
<tr>
<td>BreakpointDataTypeStr</td>
<td>Breakpoint &gt; Data Type</td>
<td>'Inherit: Same as input'</td>
</tr>
<tr>
<td>BreakpointMin</td>
<td>Breakpoint &gt; Minimum</td>
<td>'[]'</td>
</tr>
<tr>
<td>BreakpointMax</td>
<td>Breakpoint &gt; Maximum</td>
<td>'[]'</td>
</tr>
<tr>
<td>IndexDataTypeStr</td>
<td>Index &gt; Data Type</td>
<td>'int8'</td>
</tr>
<tr>
<td>FractionDataTypeStr</td>
<td>Fraction &gt; Data Type</td>
<td>'Inherit: Inherit via internal rule'</td>
</tr>
<tr>
<td>LockScale</td>
<td>Lock output data type setting against changes by the fixed-point tools</td>
<td>'off'</td>
</tr>
<tr>
<td>RndMeth</td>
<td>Integer rounding mode</td>
<td>'Ceiling'</td>
</tr>
<tr>
<td>ProcessOutOfRangeInput</td>
<td>Deprecated in R2011a</td>
<td></td>
</tr>
<tr>
<td>Sine (Sine) (masked subsystem)</td>
<td>Formula</td>
<td>'sin(2<em>pi</em>u)'</td>
</tr>
<tr>
<td>Block (Type)/Parameter</td>
<td>Dialog Box Prompt</td>
<td>Values</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>--------------------------------------------------------</td>
<td>-------------------------</td>
</tr>
<tr>
<td>NumDataPoints</td>
<td>Number of data points for lookup table</td>
<td>(\left(2^5\right)+1)</td>
</tr>
<tr>
<td>OutputWordLength</td>
<td>Output word length</td>
<td>('16')</td>
</tr>
<tr>
<td>InternalRulePriority</td>
<td>Internal rule priority for lookup table</td>
<td>{'Speed'}</td>
</tr>
</tbody>
</table>
## Math Operations Library Block Parameters

<table>
<thead>
<tr>
<th>Block (Type)/Parameter</th>
<th>Dialog Box Prompt</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abs (Abs)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ZeroCross</td>
<td>Enable zero-crossing detection</td>
<td>'off'</td>
</tr>
<tr>
<td>SampleTime</td>
<td>Sample time (-1 for inherited)</td>
<td>{'-1'}</td>
</tr>
<tr>
<td>OutMin</td>
<td>Output minimum</td>
<td>{'[]'}</td>
</tr>
<tr>
<td>OutMax</td>
<td>Output maximum</td>
<td>{'[]'}</td>
</tr>
<tr>
<td>OutDataTypeStr</td>
<td>Output data type</td>
<td>'Inherit: Inherit via internal rule'</td>
</tr>
<tr>
<td>LockScale</td>
<td>Lock output data type setting against changes by the fixed-point tools</td>
<td>{'off'}</td>
</tr>
<tr>
<td>RndMeth</td>
<td>Integer rounding mode</td>
<td>'Ceiling'</td>
</tr>
<tr>
<td>SaturateOnInteger</td>
<td>Saturate on integer overflow</td>
<td>{'off'}</td>
</tr>
<tr>
<td>Overflow</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add (Sum)</td>
<td>Icon shape</td>
<td>{'rectangular'}</td>
</tr>
<tr>
<td>IconShape</td>
<td>List of signs</td>
<td>{'++'}</td>
</tr>
<tr>
<td>Block (Type)/Parameter</td>
<td>Dialog Box Prompt</td>
<td>Values</td>
</tr>
<tr>
<td>------------------------</td>
<td>-------------------</td>
<td>--------</td>
</tr>
<tr>
<td>CollapseMode</td>
<td>Sum over</td>
<td>{'All dimensions'}</td>
</tr>
<tr>
<td>CollapseDim</td>
<td>Dimension</td>
<td>{'1'}</td>
</tr>
<tr>
<td>SampleTime</td>
<td>Sample time (-1 for inherited)</td>
<td>{'-1'}</td>
</tr>
<tr>
<td>InputSameDT</td>
<td>Require all inputs to have the same data type</td>
<td>{'off'}</td>
</tr>
<tr>
<td>AccumDataTypeStr</td>
<td>Accumulator data type</td>
<td>{'Inherit: Inherit via internal rule'}</td>
</tr>
<tr>
<td>OutMin</td>
<td>Output minimum</td>
<td>{'[]'}</td>
</tr>
<tr>
<td>OutMax</td>
<td>Output maximum</td>
<td>{'[]'}</td>
</tr>
<tr>
<td>OutDataTypeStr</td>
<td>Output data type</td>
<td>{'Inherit: Inherit via internal rule'}</td>
</tr>
<tr>
<td>LockScale</td>
<td>Lock data type settings against changes by the fixed-point tools</td>
<td>{'off'}</td>
</tr>
<tr>
<td>Block (Type)/Parameter</td>
<td>Dialog Box Prompt</td>
<td>Values</td>
</tr>
<tr>
<td>------------------------</td>
<td>----------------------------------------</td>
<td>------------------------------------------------------------------------</td>
</tr>
<tr>
<td>RndMeth</td>
<td>Integer rounding mode</td>
<td>'Ceiling'</td>
</tr>
<tr>
<td>SaturateOnInteger</td>
<td>Saturate on integer overflow</td>
<td>{'off'}</td>
</tr>
<tr>
<td></td>
<td>Algebraic Constraint (Algebraic Constraint)</td>
<td></td>
</tr>
<tr>
<td>Constraint</td>
<td>Constraint on input signal</td>
<td>{'f(z) = 0'}</td>
</tr>
<tr>
<td>Solver</td>
<td>Algebraic Loop Solver</td>
<td>{'auto'}</td>
</tr>
<tr>
<td>Tolerance</td>
<td>Solver Tolerance</td>
<td>{'auto'}</td>
</tr>
<tr>
<td>InitialGuess</td>
<td>Initial guess</td>
<td>{'0'}</td>
</tr>
<tr>
<td></td>
<td>Assignment (Assignment)</td>
<td></td>
</tr>
<tr>
<td>NumberOfDimensions</td>
<td>Number of output dimensions</td>
<td>{'1'}</td>
</tr>
<tr>
<td>IndexMode</td>
<td>Index mode</td>
<td>'Zero-based'</td>
</tr>
<tr>
<td>OutputInitialize</td>
<td>Initialize output (Y)</td>
<td>{'Initialize using input port &lt;Y0&gt;'}</td>
</tr>
<tr>
<td>IndexOptionArray</td>
<td>Index Option</td>
<td>'Assign all'</td>
</tr>
<tr>
<td>IndexParamArray</td>
<td>Index</td>
<td>cell array</td>
</tr>
<tr>
<td>OutputSizeArray</td>
<td>Output Size</td>
<td>cell array</td>
</tr>
<tr>
<td>DiagnosticForDimensions</td>
<td>Action if any output element is not assigned</td>
<td>'Error'</td>
</tr>
<tr>
<td>SampleTime</td>
<td>Sample time (-1 for inherited)</td>
<td>{'-1'}</td>
</tr>
<tr>
<td>Block (Type)/Parameter</td>
<td>Dialog Box Prompt</td>
<td>Values</td>
</tr>
<tr>
<td>------------------------</td>
<td>-------------------</td>
<td>--------</td>
</tr>
<tr>
<td>IndexOptions</td>
<td>See the IndexOptionArray parameter for more information.</td>
<td></td>
</tr>
<tr>
<td>Indices</td>
<td>See the IndexParamArray parameter for more information.</td>
<td></td>
</tr>
<tr>
<td>OutputSizes</td>
<td>See the OutputSizeArray parameter for more information.</td>
<td></td>
</tr>
<tr>
<td>Bias (Bias)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bias</td>
<td>Bias</td>
<td>{'0.0'}</td>
</tr>
<tr>
<td>SaturateOnInteger</td>
<td>Saturate on integer overflow</td>
<td>{'off'}</td>
</tr>
<tr>
<td>Overflow</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Complex to Magnitude-Angle (ComplexToMagnitudeAngle)</td>
<td>Output</td>
<td>Output</td>
</tr>
<tr>
<td>SampleTime</td>
<td>Sample time (-1 for inherited)</td>
<td>{'-1'}</td>
</tr>
<tr>
<td>Complex to Real-Imag (ComplexToRealImag)</td>
<td>Output</td>
<td>Output</td>
</tr>
<tr>
<td>SampleTime</td>
<td>Sample time (-1 for inherited)</td>
<td>{'-1'}</td>
</tr>
<tr>
<td>Divide (Product)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inputs</td>
<td>Number of inputs</td>
<td>{'*/'}</td>
</tr>
<tr>
<td>Multiplication</td>
<td>Multiplication</td>
<td>{'Element-wise(.*)'}</td>
</tr>
<tr>
<td>SampleTime</td>
<td>Sample time (-1 for inherited)</td>
<td>{'-1'}</td>
</tr>
<tr>
<td>InputSameDT</td>
<td>Require all inputs to have same data type</td>
<td>{'off'}</td>
</tr>
<tr>
<td>OutMin</td>
<td>Output minimum</td>
<td>{'[]'}</td>
</tr>
<tr>
<td>OutMax</td>
<td>Output maximum</td>
<td>{'[]'}</td>
</tr>
<tr>
<td>Block (Type)/Parameter</td>
<td>Dialog Box Prompt</td>
<td>Values</td>
</tr>
<tr>
<td>-----------------------</td>
<td>-------------------</td>
<td>--------</td>
</tr>
<tr>
<td>OutDataTypeStr</td>
<td>Output data type</td>
<td>{'Inherit: Inherit via internal rule'}</td>
</tr>
<tr>
<td>LockScale</td>
<td>Lock output data type setting against changes by the fixed-point tools</td>
<td>{'off'}</td>
</tr>
<tr>
<td>RndMeth</td>
<td>Integer rounding mode</td>
<td>'Ceiling'</td>
</tr>
<tr>
<td>SaturateOnIntegerOverflow</td>
<td>Saturate on integer overflow</td>
<td>{'off'}</td>
</tr>
</tbody>
</table>

**Dot Product (DotProduct)**

<table>
<thead>
<tr>
<th>Block (Type)/Parameter</th>
<th>Dialog Box Prompt</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>SampleTime</td>
<td>Sample time (-1 for inherited)</td>
<td>{'-1'}</td>
</tr>
<tr>
<td>InputSameDT</td>
<td>Require all inputs to have same data type</td>
<td>'off'</td>
</tr>
<tr>
<td>OutMin</td>
<td>Output minimum</td>
<td>{'[]'}</td>
</tr>
<tr>
<td>OutMax</td>
<td>Output maximum</td>
<td>{'[]'}</td>
</tr>
<tr>
<td>Block (Type)/Parameter</td>
<td>Dialog Box Prompt</td>
<td>Values</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>---------------------------------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>OutDataTypeStr</td>
<td>Output data type</td>
<td>{'Inherit: Inherit via internal rule'}</td>
</tr>
<tr>
<td>OutputDataTypeScaling Mode</td>
<td>Deprecated in R2007b</td>
<td></td>
</tr>
<tr>
<td>OutDataType</td>
<td>Deprecated in R2007b</td>
<td></td>
</tr>
<tr>
<td>OutScaling</td>
<td>Deprecated in R2007b</td>
<td></td>
</tr>
<tr>
<td>LockScale</td>
<td>Lock output data type setting against changes by the fixed-point tools</td>
<td>{'off'}</td>
</tr>
<tr>
<td>RndMeth</td>
<td>Integer rounding mode</td>
<td>'Ceiling'</td>
</tr>
<tr>
<td>SaturateOnInteger Overflow</td>
<td>Saturate on integer overflow</td>
<td>{'off'}</td>
</tr>
<tr>
<td>Find (Find)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IndexOutputFormat</td>
<td>Index output format</td>
<td>{'Linear indices'}</td>
</tr>
<tr>
<td>NumberOfInputDimensions</td>
<td>Number of input dimensions</td>
<td>integer — {'1'}</td>
</tr>
<tr>
<td>IndexMode</td>
<td>Index mode</td>
<td>{'Zero-based'}</td>
</tr>
<tr>
<td>Block (Type)/Parameter</td>
<td>Dialog Box Prompt</td>
<td>Values</td>
</tr>
<tr>
<td>------------------------</td>
<td>-------------------</td>
<td>--------</td>
</tr>
<tr>
<td>ShowOutputForNonzeroInputValues</td>
<td>Show output port for nonzero input values</td>
<td>{'off'}</td>
</tr>
<tr>
<td>SampleTime</td>
<td>Sample time (-1 for inherited)</td>
<td>{'-1'}</td>
</tr>
<tr>
<td>OutDataTypeStr</td>
<td>Output data type</td>
<td>{'Inherit: Inherit via internal rule'}</td>
</tr>
<tr>
<td>Gain (Gain)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Gain</td>
<td>Gain</td>
<td>{'1'}</td>
</tr>
</tbody>
</table>
| Multiplication | Multiplication | {'Element-wise(K.*u)'} | 'Matrix(K*u)' | 'Matrix(u*K)' | 'Matrix(K*u) (u vector)'
| SampleTime | Sample time (-1 for inherited) | {'-1'} |
| OutMin | Output minimum | {'[]'} |
| OutMax | Output maximum | {'[]'} |
| OutDataTypeStr | Output data type | {'Inherit: Inherit via internal rule'} | 'Inherit: Inherit via back propagation' | 'Inherit: Same as input' | 'double' | 'single' | 'int8' | 'uint8' | 'int16' | 'uint16' | 'int32' | 'uint32' | 'int64' | 'uint64' | 'fixdt(1,16,0)' | 'fixdt(1,16,2^0,0)'
<p>| LockScale | Lock output data type setting against changes by the fixed-point tools | {'off'} | 'on' |</p>
<table>
<thead>
<tr>
<th>Block (Type)/Parameter</th>
<th>Dialog Box Prompt</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>RndMeth</td>
<td>Integer rounding mode</td>
<td>'Ceiling'</td>
</tr>
<tr>
<td>SaturateOnInteger Overflow</td>
<td>Saturate on integer overflow</td>
<td>{'off'}</td>
</tr>
<tr>
<td>ParamMin</td>
<td>Parameter minimum</td>
<td>{'[]'}</td>
</tr>
<tr>
<td>ParamMax</td>
<td>Parameter maximum</td>
<td>{'[]'}</td>
</tr>
<tr>
<td>ParamDataTypeStr</td>
<td>Parameter data type</td>
<td>{'Inherit: Inherit via internal rule'}</td>
</tr>
</tbody>
</table>

Magnitude-Angle to Complex (MagnitudeAngleToComplex)

<table>
<thead>
<tr>
<th>Input</th>
<th>Input</th>
<th>'Magnitude'</th>
<th>'Angle'</th>
<th>{'Magnitude and angle'}</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConstantPart</td>
<td>Magnitude or Angle</td>
<td>{'0'}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ApproximationMethod</td>
<td>Approximation method</td>
<td>{'None'}</td>
<td>'CORDIC'</td>
<td></td>
</tr>
<tr>
<td>NumberOfIterations</td>
<td>Number of iterations</td>
<td>{'11'}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ScaleReciprocalGainFactor</td>
<td>Scale output by reciprocal of gain factor</td>
<td>'off'</td>
<td>{'on'}</td>
<td></td>
</tr>
<tr>
<td>SampleTime</td>
<td>Sample time (-1 for inherited)</td>
<td>{'-1'}</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Math Function (Math)
<table>
<thead>
<tr>
<th>Block (Type)/Parameter</th>
<th>Dialog Box Prompt</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operator</td>
<td>Function</td>
<td>{ 'exp' }</td>
</tr>
<tr>
<td>OutputSignalType</td>
<td>Output signal type</td>
<td>{ 'auto' }</td>
</tr>
<tr>
<td>SampleTime</td>
<td>Sample time (-1 for inherited)</td>
<td>{ '-1' }</td>
</tr>
<tr>
<td>OutMin</td>
<td>Output minimum</td>
<td>{ '[]' }</td>
</tr>
<tr>
<td>OutMax</td>
<td>Output maximum</td>
<td>{ '[]' }</td>
</tr>
<tr>
<td>OutDataTypeStr</td>
<td>Output data type</td>
<td>{ 'Inherit: Inherit via internal rule'</td>
</tr>
<tr>
<td>LockScale</td>
<td>Lock output data type setting against changes by the fixed-point tools</td>
<td>{ 'off' }</td>
</tr>
<tr>
<td>RndMeth</td>
<td>Integer rounding mode</td>
<td>{ 'Ceiling'</td>
</tr>
<tr>
<td>SaturateOnInteger Overflow</td>
<td>Saturate on integer overflow</td>
<td>{ 'off' }</td>
</tr>
<tr>
<td>Block (Type)/Parameter</td>
<td>Dialog Box Prompt</td>
<td>Values</td>
</tr>
<tr>
<td>------------------------------</td>
<td>----------------------------</td>
<td>-----------------------------------------</td>
</tr>
<tr>
<td>Matrix Concatenate (Concatenate)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NumInputs</td>
<td>Number of inputs</td>
<td>{'2'}</td>
</tr>
<tr>
<td>Mode</td>
<td>Mode</td>
<td>'Vector'</td>
</tr>
<tr>
<td>ConcatenateDimension</td>
<td>Concatenate dimension</td>
<td>{'2'}</td>
</tr>
<tr>
<td>MinMax (MinMax)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Function</td>
<td>Function</td>
<td>{'min'}</td>
</tr>
<tr>
<td>Inputs</td>
<td>Number of input ports</td>
<td>{'1'}</td>
</tr>
<tr>
<td>ZeroCross</td>
<td>Enable zero-crossing detection</td>
<td>'off'</td>
</tr>
<tr>
<td>SampleTime</td>
<td>Sample time (-1 for inherited)</td>
<td>{'-1'}</td>
</tr>
<tr>
<td>InputSameDT</td>
<td>Require all inputs to have the same data type</td>
<td>{'off'}</td>
</tr>
<tr>
<td>OutMin</td>
<td>Output minimum</td>
<td>{'[]'}</td>
</tr>
<tr>
<td>OutMax</td>
<td>Output maximum</td>
<td>{'[]'}</td>
</tr>
<tr>
<td>OutDataTypeStr</td>
<td>Output data type</td>
<td>{'Inherit: Inherit via internal rule'}</td>
</tr>
<tr>
<td>LockScale</td>
<td>Lock output data type setting against changes by the fixed-point tools</td>
<td>{'off'}</td>
</tr>
<tr>
<td>RndMeth</td>
<td>Integer rounding mode</td>
<td>'Ceiling'</td>
</tr>
<tr>
<td>Block (Type)/Parameter</td>
<td>Dialog Box Prompt</td>
<td>Values</td>
</tr>
<tr>
<td>----------------------------------------</td>
<td>------------------------------------</td>
<td>-------------------------------</td>
</tr>
<tr>
<td>SaturateOnInteger Overflow</td>
<td>Saturate on integer overflow</td>
<td>{'off'}</td>
</tr>
<tr>
<td>MinMax Running Resettable (MinMax Running Resettable) (masked subsystem)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Function</td>
<td>Function</td>
<td>{'min'}</td>
</tr>
<tr>
<td>vinit</td>
<td>Initial condition</td>
<td>{'0.0'}</td>
</tr>
<tr>
<td>Permute Dimensions (PermuteDimensions)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Order</td>
<td>Order</td>
<td>{'[2,1]'}</td>
</tr>
<tr>
<td>Polynomial (Polyval)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>coefs</td>
<td>Polynomial Coefficients</td>
<td>{'[ +2.081618890e-019, -1.441693666e-014, +4.719686976e-010, -8.536869453e-006, +1.621573104e-001, -8.087801117e+001 ]' }</td>
</tr>
<tr>
<td>Product (Product)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inputs</td>
<td>Number of inputs</td>
<td>{'2'}</td>
</tr>
<tr>
<td>Multiplication</td>
<td>Multiplication</td>
<td>{'Element-wise(.)')</td>
</tr>
<tr>
<td>CollapseMode</td>
<td>Multiply over</td>
<td>{'All dimensions'}</td>
</tr>
<tr>
<td>CollapseDim</td>
<td>Dimension</td>
<td>{'1'}</td>
</tr>
<tr>
<td>SampleTime</td>
<td>Sample time (-1 for inherited)</td>
<td>{'-1'}</td>
</tr>
<tr>
<td>InputSameDT</td>
<td>Require all inputs to have same data type</td>
<td>{'off'}</td>
</tr>
<tr>
<td>OutMin</td>
<td>Output minimum</td>
<td>{'[]}</td>
</tr>
<tr>
<td>OutMax</td>
<td>Output maximum</td>
<td>{'[]}</td>
</tr>
<tr>
<td>Block (Type)/Parameter</td>
<td>Dialog Box Prompt</td>
<td>Values</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>------------------------------------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>OutDataTypeStr</td>
<td>Output data type</td>
<td>{'Inherit: Inherit via internal rule'}</td>
</tr>
<tr>
<td>LockScale</td>
<td>Lock output data type setting against changes by the fixed-point tools</td>
<td>{'off'}</td>
</tr>
<tr>
<td>RndMeth</td>
<td>Integer rounding mode</td>
<td>'Ceiling'</td>
</tr>
<tr>
<td>SaturateOnInteger Overflow</td>
<td>Saturate on integer overflow</td>
<td>{'off'}</td>
</tr>
<tr>
<td></td>
<td>Product of Elements (Product)</td>
<td>Internals</td>
</tr>
<tr>
<td>Inputs</td>
<td>Number of inputs</td>
<td>{'*'}</td>
</tr>
<tr>
<td>Multiplication</td>
<td>Multiplication</td>
<td>{'Element-wise(.*).'}</td>
</tr>
<tr>
<td>CollapseMode</td>
<td>Multiply over</td>
<td>{'All dimensions'}</td>
</tr>
<tr>
<td>CollapseDim</td>
<td>Dimension</td>
<td>{'1'}</td>
</tr>
<tr>
<td>SampleTime</td>
<td>Sample time (-1 for inherited)</td>
<td>{'-1'}</td>
</tr>
<tr>
<td>InputSameDT</td>
<td>Require all inputs to have same data type</td>
<td>{'off'}</td>
</tr>
<tr>
<td>OutMin</td>
<td>Output minimum</td>
<td>{'[]}</td>
</tr>
<tr>
<td>Block (Type)/Parameter</td>
<td>Dialog Box Prompt</td>
<td>Values</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>-----------------------------------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>OutMax</td>
<td>Output maximum</td>
<td>{'[]'}</td>
</tr>
<tr>
<td>OutDataTypeStr</td>
<td>Output data type</td>
<td>{'Inherit: Inherit via internal rule'}</td>
</tr>
<tr>
<td>LockScale</td>
<td>Lock output data type setting</td>
<td>{'off'}</td>
</tr>
<tr>
<td></td>
<td>against changes by the fixed-point tools</td>
<td></td>
</tr>
<tr>
<td>RndMeth</td>
<td>Integer rounding mode</td>
<td>'Ceiling'</td>
</tr>
<tr>
<td>SaturateOnInteger Overflow</td>
<td>Saturate on integer overflow</td>
<td>{'off'}</td>
</tr>
<tr>
<td>Real-Imag to Complex (RealImagToComplex)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Input</td>
<td>Input</td>
<td>'Real'</td>
</tr>
<tr>
<td>ConstantPart</td>
<td>Real part or Imag part</td>
<td>{'0'}</td>
</tr>
<tr>
<td>SampleTime</td>
<td>Sample time (-1 for inherited)</td>
<td>{'-1'}</td>
</tr>
<tr>
<td>Reciprocal Sqrt (Sqrt)</td>
<td>Function</td>
<td>'sqrt'</td>
</tr>
<tr>
<td>OutputSignalType</td>
<td>Output signal type</td>
<td>{'auto'}</td>
</tr>
<tr>
<td>Block (Type)/Parameter</td>
<td>Dialog Box Prompt</td>
<td>Values</td>
</tr>
<tr>
<td>----------------------------</td>
<td>--------------------------------------------</td>
<td>---------------------------------------------</td>
</tr>
<tr>
<td>SampleTime</td>
<td>Sample time (-1 for inherited)</td>
<td>{'-1'}</td>
</tr>
<tr>
<td>OutMin</td>
<td>Output minimum</td>
<td>{'[]'}</td>
</tr>
<tr>
<td>OutMax</td>
<td>Output maximum</td>
<td>{'[]'}</td>
</tr>
<tr>
<td>OutDataTypeStr</td>
<td>Output data type</td>
<td>{'Inherit: Inherit via internal rule'</td>
</tr>
<tr>
<td>LockScale</td>
<td>Lock output data type setting against changes by the fixed-point tools</td>
<td>{'off'}</td>
</tr>
<tr>
<td>RndMeth</td>
<td>Integer rounding mode</td>
<td>{'Ceiling'</td>
</tr>
<tr>
<td>SaturateOnInteger Overflow</td>
<td>Saturate on integer overflow</td>
<td>{'off'}</td>
</tr>
<tr>
<td>Block (Type)/Parameter</td>
<td>Dialog Box Prompt</td>
<td>Values</td>
</tr>
<tr>
<td>------------------------</td>
<td>------------------</td>
<td>--------</td>
</tr>
<tr>
<td>IntermediateResults</td>
<td>Intermediate results data type</td>
<td>{'Inherit: Inherit via internal rule'}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>'Inherit: Inherit from input'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>'Inherit: Inherit from output'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>'double'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>'single'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>'int8'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>'uint8'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>'int16'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>'uint16'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>'int32'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>'uint32'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>'fixdt(1,16,0)'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>'fixdt(1,16,2^0,0)'</td>
</tr>
<tr>
<td>AlgorithmType</td>
<td>Method</td>
<td>'Exact'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{'Newton-Raphson'}</td>
</tr>
<tr>
<td>Iterations</td>
<td>Number of iterations</td>
<td>{'3'}</td>
</tr>
<tr>
<td>Reshape (Reshape)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OutputDimensionality</td>
<td>Output dimensionality</td>
<td>{'1-D array'}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>'Column vector (2-D)'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>'Row vector (2-D)'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>'Customize'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>'Derive from reference input port'</td>
</tr>
<tr>
<td>OutputDimensions</td>
<td>Output dimensions</td>
<td>{'[1,1]'}</td>
</tr>
<tr>
<td>Rounding Function</td>
<td>Function</td>
<td>{'floor'}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>'ceil'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>'round'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>'fix'</td>
</tr>
<tr>
<td>SampleTime</td>
<td>Sample time (-1 for inherited)</td>
<td>{'-1'}</td>
</tr>
<tr>
<td>Sign (Signum)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ZeroCross</td>
<td>Enable zero-crossing detection</td>
<td>'off'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{'on'}</td>
</tr>
<tr>
<td>SampleTime</td>
<td>Sample time (-1 for inherited)</td>
<td>{'-1'}</td>
</tr>
<tr>
<td>Signed Sqrt (Sqrt)</td>
<td>Function</td>
<td>'sqrt'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{'signedSqrt'}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>'rSqrt'</td>
</tr>
<tr>
<td>Block (Type)/Parameter</td>
<td>Dialog Box Prompt</td>
<td>Values</td>
</tr>
<tr>
<td>--------------------------</td>
<td>---------------------------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>OutputSignalType</td>
<td>Output signal type</td>
<td>{'auto'}</td>
</tr>
<tr>
<td>SampleTime</td>
<td>Sample time (-1 for inherited)</td>
<td>{'-1'}</td>
</tr>
<tr>
<td>OutMin</td>
<td>Output minimum</td>
<td>{'[]'}</td>
</tr>
<tr>
<td>OutMax</td>
<td>Output maximum</td>
<td>{'[]'}</td>
</tr>
<tr>
<td>OutDataTypeStr</td>
<td>Output data type</td>
<td>'Inherit: Inherit via internal rule'</td>
</tr>
<tr>
<td>LockScale</td>
<td>Lock output data type setting against changes by the fixed-point tools</td>
<td>{'off'}</td>
</tr>
<tr>
<td>RndMeth</td>
<td>Integer rounding mode</td>
<td>'Ceiling'</td>
</tr>
<tr>
<td>SaturateOnInteger Overflow</td>
<td>Saturate on integer overflow</td>
<td>'off'</td>
</tr>
<tr>
<td>Block (Type)/Parameter</td>
<td>Dialog Box Prompt</td>
<td>Values</td>
</tr>
<tr>
<td>------------------------</td>
<td>-------------------</td>
<td>--------</td>
</tr>
<tr>
<td>IntermediateResults</td>
<td>Intermediate results data type</td>
<td>{'Inherit: Inherit via internal rule'}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>'Inherit: Inherit from input'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>'Inherit: Inherit from output'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>'double'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>'single'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>'int8'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>'uint8'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>'int16'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>'uint16'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>'int32'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>'uint32'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>'fixdt(1,16,0)'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>'fixdt(1,16,2^0,0)'</td>
</tr>
<tr>
<td>AlgorithmType</td>
<td>Method</td>
<td>{'Exact'}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>'Newton-Raphson'</td>
</tr>
<tr>
<td>Iterations</td>
<td>Number of iterations</td>
<td>{'3'}</td>
</tr>
<tr>
<td>Sine Wave Function (Sin)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SineType</td>
<td>Sine type</td>
<td>{'Time based'}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>'Sample based'</td>
</tr>
<tr>
<td>TimeSource</td>
<td>Time</td>
<td>'Use simulation time'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{'Use external signal'}</td>
</tr>
<tr>
<td>Amplitude</td>
<td>Amplitude</td>
<td>{'1'}</td>
</tr>
<tr>
<td>Bias</td>
<td>Bias</td>
<td>{'0'}</td>
</tr>
<tr>
<td>Frequency</td>
<td>Frequency</td>
<td>{'1'}</td>
</tr>
<tr>
<td>Phase</td>
<td>Phase</td>
<td>{'0'}</td>
</tr>
<tr>
<td>Samples</td>
<td>Samples per period</td>
<td>{'10'}</td>
</tr>
<tr>
<td>Offset</td>
<td>Number of offset samples</td>
<td>{'0'}</td>
</tr>
<tr>
<td>SampleTime</td>
<td>Sample time</td>
<td>{'0'}</td>
</tr>
<tr>
<td>VectorParams1D</td>
<td>Interpret vector parameters as 1-D</td>
<td>'off'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{'on'}</td>
</tr>
<tr>
<td>Slider Gain (Slider Gain) (masked subsystem)</td>
<td>Low</td>
<td>{'0'}</td>
</tr>
<tr>
<td>low</td>
<td></td>
<td></td>
</tr>
<tr>
<td>gain</td>
<td>Gain</td>
<td>{'1'}</td>
</tr>
<tr>
<td>high</td>
<td>High</td>
<td>{'2'}</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Block (Type)/Parameter</td>
<td>Dialog Box Prompt</td>
<td>Values</td>
</tr>
<tr>
<td>------------------------</td>
<td>-------------------</td>
<td>--------</td>
</tr>
<tr>
<td>Sqrt (Sq rt)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Operator</td>
<td>Function</td>
<td>{'sqrt'}</td>
</tr>
<tr>
<td>OutputSignalType</td>
<td>Output signal type</td>
<td>{'auto'}</td>
</tr>
<tr>
<td>SampleTime</td>
<td>Sample time (-1 for inherited)</td>
<td>{'-1'}</td>
</tr>
<tr>
<td>OutMin</td>
<td>Output minimum</td>
<td>{'[]'}</td>
</tr>
<tr>
<td>OutMax</td>
<td>Output maximum</td>
<td>{'[]'}</td>
</tr>
<tr>
<td>OutDataTypeStr</td>
<td>Output data type</td>
<td>'Inherit: Inherit via internal rule'</td>
</tr>
<tr>
<td>LockScale</td>
<td>Lock output data type setting against changes by the fixed-point tools</td>
<td>{'off'}</td>
</tr>
<tr>
<td>RndMeth</td>
<td>Integer rounding mode</td>
<td>'Ceiling'</td>
</tr>
<tr>
<td>SaturateOnInteger</td>
<td>Saturate on integer overflow</td>
<td>'off'</td>
</tr>
<tr>
<td>Block (Type)/Parameter</td>
<td>Dialog Box Prompt</td>
<td>Values</td>
</tr>
<tr>
<td>------------------------</td>
<td>------------------</td>
<td>--------</td>
</tr>
<tr>
<td>IntermediateResults</td>
<td>Intermediate results data type</td>
<td>{'Inherit: Inherit via internal rule'}</td>
</tr>
<tr>
<td>DataTypeStr</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AlgorithmType</td>
<td>Method</td>
<td>{'Exact'}</td>
</tr>
<tr>
<td>Iterations</td>
<td>Number of iterations</td>
<td>{'3'}</td>
</tr>
<tr>
<td>Squeeze (Squeeze)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>None</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>Subtract (Sum)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IconShape</td>
<td>Icon shape</td>
<td>{'rectangular'}</td>
</tr>
<tr>
<td>Inputs</td>
<td>List of signs</td>
<td>{'+-'}</td>
</tr>
<tr>
<td>CollapseMode</td>
<td>Sum over</td>
<td>{'All dimensions'}</td>
</tr>
<tr>
<td>CollapseDim</td>
<td>Dimension</td>
<td>{'1'}</td>
</tr>
<tr>
<td>SampleTime</td>
<td>Sample time (-1 for inherited)</td>
<td>{'-1'}</td>
</tr>
<tr>
<td>InputSameDT</td>
<td>Require all inputs to have the same data type</td>
<td>{'off'}</td>
</tr>
<tr>
<td>Block (Type)/Parameter</td>
<td>Dialog Box Prompt</td>
<td>Values</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>------------------------------------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>AccumDataTypeStr</td>
<td>Accumulator data type</td>
<td>{'Inherit: Inherit via internal rule'}</td>
</tr>
<tr>
<td>OutMin</td>
<td>Output minimum</td>
<td>{'[']'}</td>
</tr>
<tr>
<td>OutMax</td>
<td>Output maximum</td>
<td>{'[']'}</td>
</tr>
<tr>
<td>OutDataTypeStr</td>
<td>Output data type</td>
<td>{'Inherit: Inherit via internal rule'}</td>
</tr>
<tr>
<td>LockScale</td>
<td>Lock data type settings against</td>
<td>{'off'}</td>
</tr>
<tr>
<td></td>
<td>changes by the fixed-point tools</td>
<td></td>
</tr>
<tr>
<td>RndMeth</td>
<td>Integer rounding mode</td>
<td>'Ceiling'</td>
</tr>
<tr>
<td>SaturateOnInteger Overflow</td>
<td>Saturate on integer overflow</td>
<td>{'off'}</td>
</tr>
<tr>
<td>Block (Type)/Parameter</td>
<td>Dialog Box Prompt</td>
<td>Values</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>----------------------------</td>
<td>---------------------------------------------------------------</td>
</tr>
<tr>
<td>Sum (Sum)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IconShape</td>
<td>Icon shape</td>
<td>'rectangular'</td>
</tr>
<tr>
<td>Inputs</td>
<td>List of signs</td>
<td>{'</td>
</tr>
<tr>
<td>CollapseMode</td>
<td>Sum over</td>
<td>{'All dimensions'}</td>
</tr>
<tr>
<td>CollapseDim</td>
<td>Dimension</td>
<td>{'1'}</td>
</tr>
<tr>
<td>SampleTime</td>
<td>Sample time (-1 for inherited)</td>
<td>{'-1'}</td>
</tr>
<tr>
<td>InputSameDT</td>
<td>Require all inputs to have the same data type</td>
<td>{'off'}</td>
</tr>
<tr>
<td>AccumDataTypeStr</td>
<td>Accumulator data type</td>
<td>{'Inherit: Inherit via internal rule'}</td>
</tr>
<tr>
<td>OutMin</td>
<td>Output minimum</td>
<td>{'[]'}</td>
</tr>
<tr>
<td>OutMax</td>
<td>Output maximum</td>
<td>{'[]'}</td>
</tr>
<tr>
<td>Block (Type)/Parameter</td>
<td>Dialog Box Prompt</td>
<td>Values</td>
</tr>
<tr>
<td>--------------------------</td>
<td>---------------------------------------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>OutDataTypeStr</td>
<td>Output data type</td>
<td>{'Inherit: Inherit via internal rule'}</td>
</tr>
<tr>
<td>LockScale</td>
<td>Lock data type settings against changes by the fixed-point tools</td>
<td>{'off'}</td>
</tr>
<tr>
<td>RndMeth</td>
<td>Integer rounding mode</td>
<td>'Ceiling'</td>
</tr>
<tr>
<td>SaturateOnIntegerOverflow</td>
<td>Saturate on integer overflow</td>
<td>{'off'}</td>
</tr>
<tr>
<td>Sum of Elements (Sum)</td>
<td>Icon shape</td>
<td>{'rectangular'}</td>
</tr>
<tr>
<td>IconShape</td>
<td>List of signs</td>
<td>{'+'}</td>
</tr>
<tr>
<td>Inputs</td>
<td>Sum over</td>
<td>{'All dimensions'}</td>
</tr>
<tr>
<td>CollapseMode</td>
<td>Dimension</td>
<td>{'1'}</td>
</tr>
<tr>
<td>CollapseDim</td>
<td>Sample time (-1 for inherited)</td>
<td>{'-1'}</td>
</tr>
<tr>
<td>InputSameDT</td>
<td>Require all inputs to have the same data type</td>
<td>{'off'}</td>
</tr>
<tr>
<td>Block (Type)/Parameter</td>
<td>Dialog Box Prompt</td>
<td>Values</td>
</tr>
<tr>
<td>------------------------</td>
<td>-------------------</td>
<td>--------</td>
</tr>
<tr>
<td>AccumDataTypeStr</td>
<td>Accumulator data type</td>
<td>{'Inherit: Inherit via internal rule'}</td>
</tr>
<tr>
<td>OutMin</td>
<td>Output minimum</td>
<td>{'[]'}</td>
</tr>
<tr>
<td>OutMax</td>
<td>Output maximum</td>
<td>{'[]'}</td>
</tr>
<tr>
<td>OutDataTypeStr</td>
<td>Output data type</td>
<td>{'Inherit: Inherit via internal rule'}</td>
</tr>
<tr>
<td>LockScale</td>
<td>Lock data type settings against changes by the fixed-point tools</td>
<td>{'off'}</td>
</tr>
<tr>
<td>RndMeth</td>
<td>Integer rounding mode</td>
<td>'Ceiling'</td>
</tr>
<tr>
<td>SaturateOnInteger</td>
<td>Saturate on integer overflow</td>
<td>{'off'}</td>
</tr>
</tbody>
</table>

Trigonometric Function (Trigonometry)
<table>
<thead>
<tr>
<th>Block (Type)/Parameter</th>
<th>Dialog Box Prompt</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Operator</strong></td>
<td>Function</td>
<td>{'sin'}</td>
</tr>
<tr>
<td><strong>ApproximationMethod</strong></td>
<td>Approximation method</td>
<td>{'None'}</td>
</tr>
<tr>
<td><strong>NumberOfIterations</strong></td>
<td>Number of iterations</td>
<td>{'11'}</td>
</tr>
<tr>
<td><strong>OutputSignalType</strong></td>
<td>Output signal type</td>
<td>{'auto'}</td>
</tr>
<tr>
<td><strong>SampleTime</strong></td>
<td>Sample time (-1 for inherited)</td>
<td>{'-1'}</td>
</tr>
<tr>
<td><strong>Unary Minus (UnaryMinus)</strong></td>
<td>Sample time (-1 for inherited)</td>
<td>{'-1'}</td>
</tr>
<tr>
<td><strong>SaturateOnInteger Overflow</strong></td>
<td>Saturate on integer overflow</td>
<td>{'off'}</td>
</tr>
<tr>
<td><strong>Vector Concatenate (Concatenate)</strong></td>
<td>Number of inputs</td>
<td>{'2'}</td>
</tr>
<tr>
<td><strong>Mode</strong></td>
<td>Mode</td>
<td>{'Vector'}</td>
</tr>
<tr>
<td><strong>Weighted Sample Time Math (SampleTimeMath)</strong></td>
<td>Operation</td>
<td>{'+'}</td>
</tr>
<tr>
<td><strong>weightValue</strong></td>
<td>Weight value</td>
<td>{'1.0'}</td>
</tr>
<tr>
<td><strong>TsampMathImp</strong></td>
<td>Implement using</td>
<td>{'Online Calculations'}</td>
</tr>
<tr>
<td><strong>OutDataTypeStr</strong></td>
<td>Output data type</td>
<td>{'Inherit via internal rule'}</td>
</tr>
<tr>
<td>Block (Type)/Parameter</td>
<td>Dialog Box Prompt</td>
<td>Values</td>
</tr>
<tr>
<td>-----------------------</td>
<td>-----------------------------</td>
<td>---------------------------------------------</td>
</tr>
<tr>
<td>RndMeth</td>
<td>Integer rounding mode</td>
<td>'Ceiling'</td>
</tr>
<tr>
<td>SaturateOnInteger Overflow</td>
<td>Saturate on integer overflow</td>
<td>{'off'}</td>
</tr>
<tr>
<td>OutputDataTypeScaling Mode</td>
<td>Deprecated in R2009b</td>
<td></td>
</tr>
<tr>
<td>DoSatur</td>
<td>Deprecated in R2009b</td>
<td></td>
</tr>
</tbody>
</table>
## Model Verification Library Block Parameters

<table>
<thead>
<tr>
<th>Block (Type)/Parameter</th>
<th>Dialog Box Prompt</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assertion (Assertion)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Enabled</td>
<td>Enable assertion</td>
<td>'off'</td>
</tr>
<tr>
<td>AssertionFailFcn</td>
<td>Simulation callback when assertion fails</td>
<td>{}</td>
</tr>
<tr>
<td>StopWhenAssertionFail</td>
<td>Stop simulation when assertion fails</td>
<td>'off'</td>
</tr>
<tr>
<td>SampleTime</td>
<td>Sample time (-1 for inherited)</td>
<td>{'-1'}</td>
</tr>
<tr>
<td>Check Dynamic Gap (Checks_DGap) (masked subsystem)</td>
<td>enabled</td>
<td>Enable assertion</td>
</tr>
<tr>
<td>callback</td>
<td>Simulation callback when assertion fails (optional)</td>
<td>{}</td>
</tr>
<tr>
<td>stopWhenAssertionFail</td>
<td>Stop simulation when assertion fails</td>
<td>'off'</td>
</tr>
<tr>
<td>export</td>
<td>Output assertion signal</td>
<td>{'off'}</td>
</tr>
<tr>
<td>icon</td>
<td>Select icon type</td>
<td>{'graphic'}</td>
</tr>
<tr>
<td>Check Dynamic Range (Checks_DRange) (masked subsystem)</td>
<td>enabled</td>
<td>Enable assertion</td>
</tr>
<tr>
<td>callback</td>
<td>Simulation callback when assertion fails (optional)</td>
<td>{}</td>
</tr>
<tr>
<td>stopWhenAssertionFail</td>
<td>Stop simulation when assertion fails</td>
<td>'off'</td>
</tr>
<tr>
<td>export</td>
<td>Output assertion signal</td>
<td>{'off'}</td>
</tr>
<tr>
<td>icon</td>
<td>Select icon type</td>
<td>{'graphic'}</td>
</tr>
<tr>
<td>Check Static Gap (Checks_SGap) (masked subsystem)</td>
<td>max</td>
<td>Upper bound</td>
</tr>
<tr>
<td>max_included</td>
<td>Inclusive upper bound</td>
<td>'off'</td>
</tr>
<tr>
<td>min</td>
<td>Lower bound</td>
<td>{'0'}</td>
</tr>
<tr>
<td>min_included</td>
<td>Inclusive lower bound</td>
<td>'off'</td>
</tr>
<tr>
<td>Block (Type)/Parameter</td>
<td>Dialog Box Prompt</td>
<td>Values</td>
</tr>
<tr>
<td>------------------------</td>
<td>-------------------</td>
<td>--------</td>
</tr>
<tr>
<td>enabled</td>
<td>Enable assertion</td>
<td>'off'</td>
</tr>
</tbody>
</table>
| callback               | Simulation callback when assertion fails (optional) | {''},
| stopWhenAssertionFail  | Stop simulation when assertion fails | 'off' | {'on'} |
| export                 | Output assertion signal | {'off'} | 'on' |
| icon                   | Select icon type  | {'graphic'} | 'text' |

Check Static Range (Checks_SRange) (masked subsystem)

<table>
<thead>
<tr>
<th>Block (Type)/Parameter</th>
<th>Dialog Box Prompt</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>max</td>
<td>Upper bound</td>
<td>{'100'}</td>
</tr>
<tr>
<td>max_included</td>
<td>Inclusive upper bound</td>
<td>'off'</td>
</tr>
<tr>
<td>min</td>
<td>Lower bound</td>
<td>{'0'}</td>
</tr>
<tr>
<td>min_included</td>
<td>Inclusive lower bound</td>
<td>'off'</td>
</tr>
<tr>
<td>enabled</td>
<td>Enable assertion</td>
<td>'off'</td>
</tr>
</tbody>
</table>
| callback               | Simulation callback when assertion fails (optional) | {''},
| stopWhenAssertionFail  | Stop simulation when assertion fails | 'off' | {'on'} |
| export                 | Output assertion signal | {'off'} | 'on' |
| icon                   | Select icon type  | {'graphic'} | 'text' |

Check Discrete Gradient (Checks_Gradient) (masked subsystem)

<table>
<thead>
<tr>
<th>Block (Type)/Parameter</th>
<th>Dialog Box Prompt</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>gradient</td>
<td>Maximum gradient</td>
<td>{'1'}</td>
</tr>
<tr>
<td>enabled</td>
<td>Enable assertion</td>
<td>'off'</td>
</tr>
</tbody>
</table>
| callback               | Simulation callback when assertion fails (optional) | {''},
| stopWhenAssertionFail  | Stop simulation when assertion fails | 'off' | {'on'} |
| export                 | Output assertion signal | {'off'} | 'on' |
| icon                   | Select icon type  | {'graphic'} | 'text' |

Check Dynamic Lower Bound (Checks_DMin) (masked subsystem)
<table>
<thead>
<tr>
<th>Block (Type)/Parameter</th>
<th>Dialog Box Prompt</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enabled</td>
<td>Enable assertion</td>
<td>'off'</td>
</tr>
<tr>
<td>callback</td>
<td>Simulation callback when assertion fails (optional)</td>
<td>{''}</td>
</tr>
<tr>
<td>stopWhenAssertionFail</td>
<td>Stop simulation when assertion fails</td>
<td>'off'</td>
</tr>
<tr>
<td>export</td>
<td>Output assertion signal</td>
<td>{'off'}</td>
</tr>
<tr>
<td>icon</td>
<td>Select icon type</td>
<td>{'graphic'}</td>
</tr>
</tbody>
</table>

Check Dynamic Upper Bound (Checks_DMax) (masked subsystem)

| enabled | Enable assertion | 'off' | {'on'} |
| callback | Simulation callback when assertion fails (optional) | {''} |
| stopWhenAssertionFail | Stop simulation when assertion fails | 'off' | {'on'} |
| export | Output assertion signal | {'off'} | 'on' |
| icon | Select icon type | {'graphic'} | 'text' |

Check Input Resolution (Checks_Resolution) (masked subsystem)

| resolution | Resolution | {'1'} |
| enabled | Enable assertion | 'off' | {'on'} |
| callback | Simulation callback when assertion fails (optional) | {''} |
| stopWhenAssertionFail | Stop simulation when assertion fails | 'off' | {'on'} |
| export | Output assertion signal | {'off'} | 'on' |

Check Static Lower Bound (Checks_SMin) (masked subsystem)

<p>| min | Lower bound | {'0'} |
| min_included | Inclusive boundary | 'off' | {'on'} |
| enabled | Enable assertion | 'off' | {'on'} |
| callback | Simulation callback when assertion fails (optional) | {''} |</p>
<table>
<thead>
<tr>
<th>Block (Type)/Parameter</th>
<th>Dialog Box Prompt</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>stopWhenAssertionFail</td>
<td>Stop simulation when assertion fails</td>
<td>'off'</td>
</tr>
<tr>
<td>export</td>
<td>Output assertion signal</td>
<td>{'off'}</td>
</tr>
<tr>
<td>icon</td>
<td>Select icon type</td>
<td>{'graphic'}</td>
</tr>
</tbody>
</table>

Check Static Upper Bound (Checks_SMax) (masked subsystem)

| max                            | Upper bound                                | {'0'}                      |
| max_included                   | Inclusive boundary                         | 'off' | {'on'}                     |
| enabled                        | Enable assertion                            | 'off' | {'on'}                     |
| callback                       | Simulation callback when assertion fails (optional) | {''}                      |
| stopWhenAssertionFail          | Stop simulation when assertion fails       | 'off' | {'on'}                     |
| export                         | Output assertion signal                    | {'off'} | 'on'                       |
| icon                           | Select icon type                           | {'graphic'} | 'text'                     |
## Model-Wide Utilities Library Block Parameters

<table>
<thead>
<tr>
<th>Block (Type)/Parameter</th>
<th>Dialog Box Prompt</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block Support Table</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DocBlock</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ECoderFlag</td>
<td>Embedded Coder Flag</td>
<td>{''}</td>
</tr>
<tr>
<td>DocumentType</td>
<td>Document Type</td>
<td>{'Text'}</td>
</tr>
<tr>
<td>Model Info</td>
<td></td>
<td></td>
</tr>
<tr>
<td>InitialSaveTempField</td>
<td>InitialSaveTempField</td>
<td>{''}</td>
</tr>
<tr>
<td>InitialBlockCM</td>
<td>InitialBlockCM</td>
<td>{'None'}</td>
</tr>
<tr>
<td>BlockCM</td>
<td>BlockCM</td>
<td>{'None'}</td>
</tr>
<tr>
<td>Frame</td>
<td>Show block frame</td>
<td>'off'</td>
</tr>
<tr>
<td>SaveTempField</td>
<td>SaveTempField</td>
<td>{''}</td>
</tr>
<tr>
<td>DisplayStringWithTags</td>
<td>DisplayStringWithTags</td>
<td>{'Model Info'}</td>
</tr>
<tr>
<td>MaskDisplayString</td>
<td>MaskDisplayString</td>
<td>{'Model Info'}</td>
</tr>
<tr>
<td>HorizontalTextAlignment</td>
<td>Horizontal text alignment</td>
<td>{'Center'}</td>
</tr>
<tr>
<td>LeftAlignmentValue</td>
<td>LeftAlignmentValue</td>
<td>{'0.5'}</td>
</tr>
<tr>
<td>SourceBlockDiagram</td>
<td>SourceBlockDiagram</td>
<td>{'untitled'}</td>
</tr>
<tr>
<td>TagMaxNumber</td>
<td>TagMaxNumber</td>
<td>{'20'}</td>
</tr>
<tr>
<td>CMTag1</td>
<td>CMTag1</td>
<td>{''}</td>
</tr>
<tr>
<td>CMTag2</td>
<td>CMTag2</td>
<td>{''}</td>
</tr>
<tr>
<td>CMTag3</td>
<td>CMTag3</td>
<td>{''}</td>
</tr>
<tr>
<td>CMTag4</td>
<td>CMTag4</td>
<td>{''}</td>
</tr>
<tr>
<td>CMTag5</td>
<td>CMTag5</td>
<td>{''}</td>
</tr>
<tr>
<td>CMTag6</td>
<td>CMTag6</td>
<td>{''}</td>
</tr>
<tr>
<td>CMTag7</td>
<td>CMTag7</td>
<td>{''}</td>
</tr>
<tr>
<td>CMTag8</td>
<td>CMTag8</td>
<td>{''}</td>
</tr>
<tr>
<td>CMTag9</td>
<td>CMTag9</td>
<td>{''}</td>
</tr>
<tr>
<td>CMTag10</td>
<td>CMTag10</td>
<td>{''}</td>
</tr>
<tr>
<td>Block (Type)/Parameter</td>
<td>Dialog Box Prompt</td>
<td>Values</td>
</tr>
<tr>
<td>------------------------</td>
<td>-------------------</td>
<td>--------</td>
</tr>
<tr>
<td>CMTag11</td>
<td>CMTag11</td>
<td>{'''}</td>
</tr>
<tr>
<td>CMTag12</td>
<td>CMTag12</td>
<td>{'''}</td>
</tr>
<tr>
<td>CMTag13</td>
<td>CMTag13</td>
<td>{'''}</td>
</tr>
<tr>
<td>CMTag14</td>
<td>CMTag14</td>
<td>{'''}</td>
</tr>
<tr>
<td>CMTag15</td>
<td>CMTag15</td>
<td>{'''}</td>
</tr>
<tr>
<td>CMTag16</td>
<td>CMTag16</td>
<td>{'''}</td>
</tr>
<tr>
<td>CMTag17</td>
<td>CMTag17</td>
<td>{'''}</td>
</tr>
<tr>
<td>CMTag18</td>
<td>CMTag18</td>
<td>{'''}</td>
</tr>
<tr>
<td>CMTag19</td>
<td>CMTag19</td>
<td>{'''}</td>
</tr>
<tr>
<td>CMTag20</td>
<td>CMTag20</td>
<td>{'''}</td>
</tr>
</tbody>
</table>

**Timed-Based Linearization (Timed Linearization) (masked subsystem)**

<table>
<thead>
<tr>
<th>LinearizationTime</th>
<th>Linearization time</th>
<th>{'1'}</th>
</tr>
</thead>
<tbody>
<tr>
<td>SampleTime</td>
<td>Sample time (of linearized model)</td>
<td>{'0'}</td>
</tr>
</tbody>
</table>

**Trigger-Based Linearization (Triggered Linearization) (masked subsystem)**

<table>
<thead>
<tr>
<th>TriggerType</th>
<th>Trigger type</th>
<th>{'rising'}</th>
<th>{'falling'}</th>
<th>{'either'}</th>
<th>{'function-call'}</th>
</tr>
</thead>
<tbody>
<tr>
<td>SampleTime</td>
<td>Sample time (of linearized model)</td>
<td>{'0'}</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Ports & Subsystems Library Block Parameters

<table>
<thead>
<tr>
<th>Block (Type)/Parameter</th>
<th>Dialog Box Prompt</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Action Port</strong> (ActionPort)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>InitializeStates</strong></td>
<td>States when execution is resumed</td>
<td>{'held'}</td>
</tr>
<tr>
<td><strong>PropagateVarSize</strong></td>
<td>Propagate sizes of variable-size signals</td>
<td>{'Only when execution is resumed'}</td>
</tr>
<tr>
<td><strong>Atomic Subsystem</strong> (SubSystem)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>ShowPortLabels</strong></td>
<td>Show port labels</td>
<td>'none'</td>
</tr>
<tr>
<td></td>
<td><strong>Note</strong> The values 'off' and 'on' are for backward compatibility only and should not be used in new models or when updating existing models.</td>
<td></td>
</tr>
<tr>
<td><strong>BlockChoice</strong></td>
<td>Block choice</td>
<td>{''}</td>
</tr>
<tr>
<td><strong>TemplateBlock</strong></td>
<td>Template block</td>
<td>{''}</td>
</tr>
<tr>
<td><strong>MemberBlocks</strong></td>
<td>Member blocks</td>
<td>{''}</td>
</tr>
<tr>
<td><strong>Permissions</strong></td>
<td>Read/Write permissions</td>
<td>{'ReadWrite'}</td>
</tr>
<tr>
<td><strong>ErrorFcn</strong></td>
<td>Name of error callback function</td>
<td>{''}</td>
</tr>
<tr>
<td><strong>PermitHierarchicalResolution</strong></td>
<td>Permit hierarchical resolution</td>
<td>{'All'}</td>
</tr>
<tr>
<td><strong>TreatAsAtomicUnit</strong></td>
<td>Treat as atomic unit</td>
<td>'off'</td>
</tr>
<tr>
<td><strong>TreatAsGroupedWhenPropagatingVariantConditions</strong></td>
<td>Treat as grouped when propagating variant conditions</td>
<td>'off'</td>
</tr>
<tr>
<td><strong>MinAlgLoopOccurrences</strong></td>
<td>Minimize algebraic loop occurrences</td>
<td>{'off'}</td>
</tr>
<tr>
<td><strong>PropExecContextOutsideSubsystem</strong></td>
<td>Propagate execution context across subsystem boundary</td>
<td>{'off'}</td>
</tr>
<tr>
<td>Block (Type)/Parameter</td>
<td>Dialog Box Prompt</td>
<td>Values</td>
</tr>
<tr>
<td>------------------------</td>
<td>-------------------</td>
<td>--------</td>
</tr>
<tr>
<td>CheckFcnCallInp</td>
<td>Warn if function-call inputs are context-specific</td>
<td>{'off'}</td>
</tr>
<tr>
<td>InsideContextMsg</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SystemSampleTime</td>
<td>Sample time (-1 for inherited)</td>
<td>{'-1'}</td>
</tr>
<tr>
<td>RTWSystemCode</td>
<td>Function packaging</td>
<td>{'Auto'}</td>
</tr>
<tr>
<td>RTWFcnNameOpts</td>
<td>Function name options</td>
<td>{'Auto'}</td>
</tr>
<tr>
<td>RTWFcnName</td>
<td>Function name</td>
<td>{''}</td>
</tr>
<tr>
<td>RTWFileNameOpts</td>
<td>File name options</td>
<td>{'Auto'}</td>
</tr>
<tr>
<td>RTWFileName</td>
<td>File name (no extension)</td>
<td>{''}</td>
</tr>
<tr>
<td>FunctionInterfaceSpec</td>
<td>Function interface</td>
<td>{'void_void'}</td>
</tr>
<tr>
<td>FunctionWithSeparateData</td>
<td>“Function with separate data” on page 1-0</td>
<td>{'off'}</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Block (Type)/Parameter</td>
<td>Dialog Box Prompt</td>
<td>Values</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>----------------------------------------------------------------------------------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>RTWMemSecFuncInitTerm</td>
<td>“Memory section for initialize/terminate functions” on page 1-0</td>
<td>{'Inherit from model'}</td>
</tr>
<tr>
<td></td>
<td>This parameter requires a license for Embedded Coder software and an ERT-based system target file.</td>
<td></td>
</tr>
<tr>
<td>RTWMemSecFuncExecute</td>
<td>“Memory section for execution functions” on page 1-0</td>
<td>{'Inherit from model'}</td>
</tr>
<tr>
<td></td>
<td>This parameter requires a license for Embedded Coder software and an ERT-based system target file.</td>
<td></td>
</tr>
<tr>
<td>RTWMemSecDataConstants</td>
<td>“Memory section for constants” on page 1-0</td>
<td>{'Inherit from model'}</td>
</tr>
<tr>
<td></td>
<td>This parameter requires a license for Embedded Coder software and an ERT-based system target file.</td>
<td></td>
</tr>
<tr>
<td>RTWMemSecDataInternal</td>
<td>“Memory section for internal data” on page 1-0</td>
<td>{'Inherit from model'}</td>
</tr>
<tr>
<td></td>
<td>This parameter requires a license for Embedded Coder software and an ERT-based system target file.</td>
<td></td>
</tr>
<tr>
<td>RTWMemSecDataParameters</td>
<td>“Memory section for parameters” on page 1-0</td>
<td>{'Inherit from model'}</td>
</tr>
<tr>
<td></td>
<td>This parameter requires a license for Embedded Coder software and an ERT-based system target file.</td>
<td></td>
</tr>
<tr>
<td>Block (Type)/Parameter</td>
<td>Dialog Box Prompt</td>
<td>Values</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>------------------------------------------------------------------------------------</td>
<td>--------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>DataTypeOverride</strong></td>
<td>No dialog box prompt</td>
<td>{ 'UseLocalSettings' }</td>
</tr>
<tr>
<td></td>
<td>Specifies data type used to override fixed-point data types. Set by <strong>Data type override</strong> on the Fixed-Point Tool.</td>
<td></td>
</tr>
<tr>
<td><strong>MinMaxOverflowLogging</strong></td>
<td>No dialog box prompt</td>
<td>{ 'UseLocalSettings' }</td>
</tr>
<tr>
<td></td>
<td>Setting for fixed-point instrumentation. Set by <strong>Fixed-point instrumentation mode</strong> on the Fixed-Point Tool.</td>
<td></td>
</tr>
<tr>
<td><strong>SimViewingDevice</strong></td>
<td>No dialog box prompt</td>
<td>{ 'off' }</td>
</tr>
<tr>
<td></td>
<td>If set to 'on', designates the block as a Signal Viewing Subsystem — an atomic subsystem that encapsulates processing and viewing of signals received from the target system in External mode. For more information, see “Signal Viewing Subsystems” (Simulink Coder).</td>
<td></td>
</tr>
<tr>
<td><strong>IsSubsystemVirtual</strong></td>
<td></td>
<td>boolean — { 'off' }</td>
</tr>
<tr>
<td></td>
<td>Read-only</td>
<td></td>
</tr>
<tr>
<td><strong>Code Reuse Subsystem (SubSystem)</strong></td>
<td><strong>ShowPortLabels</strong></td>
<td>'none'</td>
</tr>
<tr>
<td></td>
<td>Show port labels</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Note</strong> The values 'off' and 'on' are for backward compatibility only and should not be used in new models or when updating existing models.</td>
<td></td>
</tr>
<tr>
<td><strong>BlockChoice</strong></td>
<td>Block choice</td>
<td>{ '' }</td>
</tr>
</tbody>
</table>
### Block (Type)/Parameter  |  Dialog Box Prompt  |  Values
---|---|---
TemplateBlock | Template block | {''}
MemberBlocks | Member blocks | {''}
Permissions | Read/Write permissions | {'ReadWrite'} | 'ReadOnly' | 'NoReadOrWrite'
ErrorFcn | Name of error callback function | {''}
PermitHierarchical Resolution | Permit hierarchical resolution | {'All'} | 'ExplicitOnly' | 'None'
TreatAsAtomicUnit | Treat as atomic unit | 'off' | {'on'}
MinAlgLoopOccurrences | Minimize algebraic loop occurrences | {'off'} | 'on'
PropExecContext OutsideSubsystem | Propagate execution context across subsystem boundary | {'off'} | 'on'
CheckFcnCallInp InsideContextMsg | Warn if function-call inputs are context-specific | {'off'} | 'on'
SystemSampleTime | Sample time (-1 for inherited) | {'-1'}
RTWSystemCode | Function packaging | 'Auto' | 'Inline' | 'Nonreusable function' | {'Reusable function'}
RTWFcnNameOpts | Function name options | 'Auto' | {'Use subsystem name'} | 'User specified'
RTWFcnName | Function name | {''}
RTWFileNameOpts | File name options | 'Auto' | {'Use subsystem name'} | 'Use function name' | 'User specified'
RTWFileName | File name (no extension) | {''}
<table>
<thead>
<tr>
<th>Block (Type)/Parameter</th>
<th>Dialog Box Prompt</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTWMemSecFuncInitTerm</td>
<td>“Memory section for initialize/terminate functions” on page 1-0</td>
<td>{'Inherit from model'}</td>
</tr>
<tr>
<td></td>
<td>This parameter requires a license for Embedded Coder software and an ERT-based system target file.</td>
<td></td>
</tr>
<tr>
<td>RTWMemSecFuncExecute</td>
<td>“Memory section for execution functions” on page 1-0</td>
<td>{'Inherit from model'}</td>
</tr>
<tr>
<td></td>
<td>This parameter requires a license for Embedded Coder software and an ERT-based system target file.</td>
<td></td>
</tr>
<tr>
<td>RTWMemSecDataConstants</td>
<td>“Memory section for constants” on page 1-0</td>
<td>{'Inherit from model'}</td>
</tr>
<tr>
<td></td>
<td>This parameter requires a license for Embedded Coder software and an ERT-based system target file.</td>
<td></td>
</tr>
<tr>
<td>RTWMemSecDataInternal</td>
<td>“Memory section for internal data” on page 1-0</td>
<td>{'Inherit from model'}</td>
</tr>
<tr>
<td></td>
<td>This parameter requires a license for Embedded Coder software and an ERT-based system target file.</td>
<td></td>
</tr>
<tr>
<td>RTWMemSecDataParameters</td>
<td>“Memory section for parameters” on page 1-0</td>
<td>{'Inherit from model'}</td>
</tr>
<tr>
<td></td>
<td>This parameter requires a license for Embedded Coder software and an ERT-based system target file.</td>
<td></td>
</tr>
<tr>
<td>Block (Type)/Parameter</td>
<td>Dialog Box Prompt</td>
<td>Values</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>-----------------------------------------------------------------------------------</td>
<td>--------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>DataTypeOverride</td>
<td>No dialog box prompt&lt;br&gt;Specifies data type used to override fixed-point data types. Set by <strong>DataTypeOverride</strong> on the Fixed-Point Tool.</td>
<td>{'UseLocalSettings'}</td>
</tr>
<tr>
<td>MinMaxOverflowLogging</td>
<td>No dialog box prompt&lt;br&gt;Setting for fixed-point instrumentation. Set by <strong>MinMaxOverflowLogging</strong> on the Fixed-Point Tool.</td>
<td>{'UseLocalSettings'}</td>
</tr>
<tr>
<td>IsSubsystemVirtual</td>
<td></td>
<td>boolean — {'off'}</td>
</tr>
<tr>
<td>Configurable Subsystem (SubSystem)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ShowPortLabels</td>
<td>Show port labels&lt;br&gt;<strong>Note</strong> The values 'off' and 'on' are for backward compatibility only and should not be used in new models or when updating existing models.</td>
<td>{'none'}</td>
</tr>
<tr>
<td>BlockChoice</td>
<td>Block choice&lt;br&gt;<strong>Note</strong> The values 'off' and 'on' are for backward compatibility only and should not be used in new models or when updating existing models.</td>
<td>{''}</td>
</tr>
<tr>
<td>TemplateBlock</td>
<td>Template block&lt;br&gt;<strong>Note</strong> The values 'off' and 'on' are for backward compatibility only and should not be used in new models or when updating existing models.</td>
<td>{'self'}</td>
</tr>
<tr>
<td>MemberBlocks</td>
<td>Member blocks&lt;br&gt;<strong>Note</strong> The values 'off' and 'on' are for backward compatibility only and should not be used in new models or when updating existing models.</td>
<td>{''}</td>
</tr>
<tr>
<td>Permissions</td>
<td>Read/Write permissions&lt;br&gt;<strong>Note</strong> The values 'off' and 'on' are for backward compatibility only and should not be used in new models or when updating existing models.</td>
<td>{'ReadWrite'}</td>
</tr>
<tr>
<td>ErrorFcn</td>
<td>Name of error callback function&lt;br&gt;<strong>Note</strong> The values 'off' and 'on' are for backward compatibility only and should not be used in new models or when updating existing models.</td>
<td>{''}</td>
</tr>
<tr>
<td>PermitHierarchicalResolution</td>
<td>Permit hierarchical resolution&lt;br&gt;<strong>Note</strong> The values 'off' and 'on' are for backward compatibility only and should not be used in new models or when updating existing models.</td>
<td>{'All'}</td>
</tr>
<tr>
<td>TreatAsAtomicUnit</td>
<td>Treat as atomic unit&lt;br&gt;<strong>Note</strong> The values 'off' and 'on' are for backward compatibility only and should not be used in new models or when updating existing models.</td>
<td>{'off'}</td>
</tr>
<tr>
<td>Block (Type)/Parameter</td>
<td>Dialog Box Prompt</td>
<td>Values</td>
</tr>
<tr>
<td>---------------------------------</td>
<td>----------------------------------------------------------------------------------</td>
<td>---------------------------------------------</td>
</tr>
<tr>
<td>MinAlgLoopOccurrences</td>
<td>Minimize algebraic loop occurrences</td>
<td>{'off'}</td>
</tr>
<tr>
<td>PropExecContext</td>
<td>Propagate execution context across subsystem boundary</td>
<td>{'off'}</td>
</tr>
<tr>
<td>PropExecContext</td>
<td>Propagate execution context across subsystem boundary</td>
<td>{'off'}</td>
</tr>
<tr>
<td>CheckFcnCallInp</td>
<td>Warn if function-call inputs are context-specific</td>
<td>{'off'}</td>
</tr>
<tr>
<td>SystemSampleTime</td>
<td>Sample time (-1 for inherited)</td>
<td>{'-1'}</td>
</tr>
<tr>
<td>RTWSystemCode</td>
<td>Function packaging</td>
<td>{'Auto'}</td>
</tr>
<tr>
<td>RTWFcnNameOpts</td>
<td>Function name options</td>
<td>{'Auto'}</td>
</tr>
<tr>
<td>RTWFcnName</td>
<td>Function name</td>
<td>{''}</td>
</tr>
<tr>
<td>RTWFileNameOpts</td>
<td>File name options</td>
<td>{'Auto'}</td>
</tr>
<tr>
<td>RTWFileName</td>
<td>File name (no extension)</td>
<td>{''}</td>
</tr>
<tr>
<td>DataTypeOverride</td>
<td>No dialog box prompt</td>
<td>{'UseLocalSettings'}</td>
</tr>
<tr>
<td>MinMaxOverflowLogging</td>
<td>Setting for fixed-point instrumentation. Set by Fixed-point instrumentation mode</td>
<td>{'UseLocalSettings'}</td>
</tr>
<tr>
<td>Block (Type)/Parameter</td>
<td>Dialog Box Prompt</td>
<td>Values</td>
</tr>
<tr>
<td>----------------------------------------</td>
<td>-----------------------------------------------------------------------------------</td>
<td>---------------------------------------------</td>
</tr>
<tr>
<td>SimViewingDevice</td>
<td>No dialog box prompt</td>
<td>{'off'}</td>
</tr>
<tr>
<td></td>
<td>If set to 'on', designates the block as a Signal Viewing Subsystem — an atomic subsystem that encapsulates processing and viewing of signals received from the target system in External mode. For more information, see “Signal Viewing Subsystems” (Simulink Coder).</td>
<td></td>
</tr>
<tr>
<td>DataTypeOverride</td>
<td>No dialog box prompt</td>
<td>{'UseLocalSettings'}</td>
</tr>
<tr>
<td></td>
<td>Specifies data type used to override fixed-point data types. Set by <strong>Data type override</strong> on the Fixed-Point Tool.</td>
<td></td>
</tr>
<tr>
<td>MinMaxOverflowLogging</td>
<td>No dialog box prompt</td>
<td>{'UseLocalSettings'}</td>
</tr>
<tr>
<td></td>
<td>Setting for fixed-point instrumentation. Set by <strong>Fixed-point instrumentation mode</strong> on the Fixed-Point Tool.</td>
<td></td>
</tr>
<tr>
<td>IsSubsystemVirtual</td>
<td>No dialog box prompt</td>
<td>boolean — {'on'}</td>
</tr>
<tr>
<td></td>
<td>Read-only</td>
<td></td>
</tr>
<tr>
<td>Enable (EnablePort)</td>
<td>States when enabling</td>
<td>{'held'}</td>
</tr>
<tr>
<td>StatesWhenEnabling</td>
<td>States when enabling</td>
<td></td>
</tr>
<tr>
<td>PropagateVarSize</td>
<td>Propagate sizes of variable-size signals</td>
<td>{'Only when enabling'}</td>
</tr>
<tr>
<td>ShowOutputPort</td>
<td>Show output port</td>
<td>{'off'}</td>
</tr>
<tr>
<td>ZeroCross</td>
<td>Enable zero-crossing detection</td>
<td>'off'</td>
</tr>
</tbody>
</table>

**Enabled and Triggered Subsystem (SubSystem)**
<table>
<thead>
<tr>
<th>Block (Type)/Parameter</th>
<th>Dialog Box Prompt</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>ShowPortLabels</td>
<td>Show port labels</td>
<td>'none'</td>
</tr>
<tr>
<td>Note</td>
<td></td>
<td>The values 'off' and 'on' are for backward compatibility only and should not be used in new models or when updating existing models.</td>
</tr>
<tr>
<td>BlockChoice</td>
<td>Block choice</td>
<td>{''}</td>
</tr>
<tr>
<td>TemplateBlock</td>
<td>Template block</td>
<td>{''}</td>
</tr>
<tr>
<td>MemberBlocks</td>
<td>Member blocks</td>
<td>{''}</td>
</tr>
<tr>
<td>Permissions</td>
<td>Read/Write permissions</td>
<td>{'ReadWrite'}</td>
</tr>
<tr>
<td>ErrorFcn</td>
<td>Name of error callback function</td>
<td>{''}</td>
</tr>
<tr>
<td>PermitHierarchicalResolution</td>
<td>Permit hierarchical resolution</td>
<td>{'All'}</td>
</tr>
<tr>
<td>TreatAsAtomicUnit</td>
<td>Treat as atomic unit</td>
<td>'off'</td>
</tr>
<tr>
<td>MinAlgLoopOccurrences</td>
<td>Minimize algebraic loop occurrences</td>
<td>{'off'}</td>
</tr>
<tr>
<td>PropExecContext</td>
<td>Propagate execution context across subsystem boundary</td>
<td>{''}</td>
</tr>
<tr>
<td>OutsideSubsystem</td>
<td></td>
<td>{''}</td>
</tr>
<tr>
<td>CheckFcnCallInp</td>
<td>Warn if function-call inputs are context-specific</td>
<td>{''}</td>
</tr>
<tr>
<td>InsideContextMsg</td>
<td></td>
<td>{''}</td>
</tr>
<tr>
<td>SystemSampleTime</td>
<td>Sample time (-1 for inherited)</td>
<td>{''}</td>
</tr>
<tr>
<td>RTWSystemCode</td>
<td>Function packaging</td>
<td>{'Auto'}</td>
</tr>
<tr>
<td>RTWFcnNameOpts</td>
<td>Function name options</td>
<td>{'Auto'}</td>
</tr>
<tr>
<td>RTWFcnName</td>
<td>Function name</td>
<td>{''}</td>
</tr>
<tr>
<td>Block (Type)/Parameter</td>
<td>Dialog Box Prompt</td>
<td>Values</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>------------------------------------------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>RTWFileNameOpts</td>
<td>File name options</td>
<td>{'Auto'}</td>
</tr>
<tr>
<td>RTWFileName</td>
<td>File name (no extension)</td>
<td>{''}</td>
</tr>
<tr>
<td>DataTypeOverride</td>
<td>No dialog box prompt</td>
<td>{'UseLocalSettings'}</td>
</tr>
<tr>
<td>MinMaxOverflowLogging</td>
<td>No dialog box prompt</td>
<td>{'UseLocalSettings'}</td>
</tr>
<tr>
<td>IsSubsystemVirtual</td>
<td></td>
<td>boolean — {'off'}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Read-only</td>
</tr>
<tr>
<td>Enabled Subsystem (SubSystem)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ShowPortLabels</td>
<td>Show port labels</td>
<td>'none'</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BlockChoice</td>
<td>Block choice</td>
<td>{''}</td>
</tr>
<tr>
<td>TemplateBlock</td>
<td>Template block</td>
<td>{''}</td>
</tr>
<tr>
<td>MemberBlocks</td>
<td>Member blocks</td>
<td>{''}</td>
</tr>
<tr>
<td>Permissions</td>
<td>Read/Write permissions</td>
<td>{'ReadWrite'}</td>
</tr>
<tr>
<td>Block (Type)/Parameter</td>
<td>Dialog Box Prompt</td>
<td>Values</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>------------------------------------------------</td>
<td>---------------------------------------------</td>
</tr>
<tr>
<td>ErrorFcn</td>
<td>Name of error callback function</td>
<td>{''}</td>
</tr>
<tr>
<td>PermitHierarchical Resolution</td>
<td>Permit hierarchical resolution</td>
<td>{'All'}</td>
</tr>
<tr>
<td>TreatAsAtomicUnit</td>
<td>Treat as atomic unit</td>
<td>'off'</td>
</tr>
<tr>
<td>MinAlgLoopOccurrences</td>
<td>Minimize algebraic loop occurrences</td>
<td>{'off'}</td>
</tr>
<tr>
<td>PropExecContext</td>
<td>Propagate execution context</td>
<td>{'off'}</td>
</tr>
<tr>
<td>OutsideSubsystem</td>
<td>Propagate execution context across subsystem boundary</td>
<td>{'off'}</td>
</tr>
<tr>
<td>CheckFcnCallInp</td>
<td>Warn if function-call inputs are context-specific</td>
<td>{'off'}</td>
</tr>
<tr>
<td>InsideContextMsg</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SystemSampleTime</td>
<td>Sample time (-1 for inherited)</td>
<td>{'-1'}</td>
</tr>
<tr>
<td>RTWSystemCode</td>
<td>Function packaging</td>
<td>{'Auto'}</td>
</tr>
<tr>
<td>RTWFcnNameOpts</td>
<td>Function name options</td>
<td>{'Auto'}</td>
</tr>
<tr>
<td>RTWFcnName</td>
<td>Function name</td>
<td>{''}</td>
</tr>
<tr>
<td>RTWFileNameOpts</td>
<td>File name options</td>
<td>{'Auto'}</td>
</tr>
<tr>
<td>RTWFileName</td>
<td>File name (no extension)</td>
<td>{''}</td>
</tr>
<tr>
<td>DataTypeOverride</td>
<td>No dialog box prompt</td>
<td>{'UseLocalSettings'}</td>
</tr>
</tbody>
</table>
### Block (Type)/Parameter

<table>
<thead>
<tr>
<th>Block (Type)/Parameter</th>
<th>Dialog Box Prompt</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>MinMaxOverflowLogging</td>
<td>No dialog box prompt</td>
<td>{'UseLocalSettings'}</td>
</tr>
<tr>
<td></td>
<td>Setting for fixed-point instrumentation. Set by <strong>Fixed-point instrumentation mode</strong> on the Fixed-Point Tool.</td>
<td></td>
</tr>
<tr>
<td>IsSubsystemVirtual</td>
<td>boolean — {'off'}</td>
<td>'on'</td>
</tr>
<tr>
<td></td>
<td>Read-only</td>
<td></td>
</tr>
<tr>
<td>For Each (ForEach)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>InputPartition</td>
<td>Partition</td>
<td>cell array of character vectors</td>
</tr>
<tr>
<td>InputPartitionDimension</td>
<td>Partition dimension for input signal</td>
<td>cell array of character vectors</td>
</tr>
<tr>
<td>InputPartitionWidth</td>
<td>Width of partition for input signal</td>
<td>cell array of character vectors</td>
</tr>
<tr>
<td>OutputConcatenationDimension</td>
<td>Concatenation dimension of output signal</td>
<td>cell array of character vectors</td>
</tr>
<tr>
<td>For Iterator (ForIterator)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ResetStates</td>
<td>States when starting</td>
<td>{'held'}</td>
</tr>
<tr>
<td>IterationSource</td>
<td>Iteration limit source</td>
<td>{'internal'}</td>
</tr>
<tr>
<td>IterationLimit</td>
<td>Iteration limit</td>
<td>{'5'}</td>
</tr>
<tr>
<td>ExternalIncrement</td>
<td>Set next i (iteration variable) externally</td>
<td>{'off'}</td>
</tr>
<tr>
<td>ShowIterationPort</td>
<td>Show iteration variable</td>
<td>'off'</td>
</tr>
<tr>
<td>IndexMode</td>
<td>Index mode</td>
<td>'Zero-based'</td>
</tr>
<tr>
<td>IterationVariableDataType</td>
<td>Iteration variable data type</td>
<td>{'int32'}</td>
</tr>
<tr>
<td>Block (Type)/Parameter</td>
<td>Dialog Box Prompt</td>
<td>Values</td>
</tr>
<tr>
<td>------------------------------</td>
<td>------------------------------------</td>
<td>------------------------------------------------------------------------</td>
</tr>
<tr>
<td>ShowPortLabels</td>
<td>Show port labels</td>
<td>'none'</td>
</tr>
<tr>
<td>BlockChoice</td>
<td>Block choice</td>
<td>{''}</td>
</tr>
<tr>
<td>TemplateBlock</td>
<td>Template block</td>
<td>{''}</td>
</tr>
<tr>
<td>MemberBlocks</td>
<td>Member blocks</td>
<td>{''}</td>
</tr>
<tr>
<td>Permissions</td>
<td>Read/Write permissions</td>
<td>{'ReadWrite'}</td>
</tr>
<tr>
<td>ErrorFcn</td>
<td>Name of error callback function</td>
<td>{''}</td>
</tr>
<tr>
<td>PermitHierarchicalResolution</td>
<td>Permit hierarchical resolution</td>
<td>{'All'}</td>
</tr>
<tr>
<td>TreatAsAtomicUnit</td>
<td>Treat as atomic unit</td>
<td>'off'</td>
</tr>
<tr>
<td>MinAlgLoopOccurrences</td>
<td>Minimize algebraic loop occurrences</td>
<td>{'off'}</td>
</tr>
<tr>
<td>PropExecContextOutsideSubsystem</td>
<td>Propagate execution context</td>
<td>{'off'}</td>
</tr>
<tr>
<td>CheckFcnCallInpInsideContextMsg</td>
<td>Warn if function-call inputs are context-specific</td>
<td>{'off'}</td>
</tr>
<tr>
<td>SystemSampleTime</td>
<td>Sample time (-1 for inherited)</td>
<td>{'-1'}</td>
</tr>
<tr>
<td>RTWSystemCode</td>
<td>Function packaging</td>
<td>{'Auto'}</td>
</tr>
<tr>
<td>RTWFcnNameOpts</td>
<td>Function name options</td>
<td>{'Auto'}</td>
</tr>
<tr>
<td>RTWFcnName</td>
<td>Function name</td>
<td>{''}</td>
</tr>
<tr>
<td>Block (Type)/Parameter</td>
<td>Dialog Box Prompt</td>
<td>Values</td>
</tr>
<tr>
<td>----------------------------</td>
<td>---------------------------------------------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>RTWFileNameOpts</td>
<td>File name options</td>
<td>{'Auto'}</td>
</tr>
<tr>
<td>RTWFileName</td>
<td>File name (no extension)</td>
<td>{''}</td>
</tr>
<tr>
<td>DataTypeOverride</td>
<td>No dialog box prompt</td>
<td>{'UseLocalSettings'}</td>
</tr>
<tr>
<td></td>
<td>Specifies data type used to override fixed-point data types. Set by <strong>Data type override</strong> on the Fixed-Point Tool.</td>
<td></td>
</tr>
<tr>
<td>MinMaxOverflowLogging</td>
<td>No dialog box prompt</td>
<td>{'UseLocalSettings'}</td>
</tr>
<tr>
<td></td>
<td>Setting for fixed-point instrumentation. Set by <strong>Fixed-point instrumentation mode</strong> on the Fixed-Point Tool.</td>
<td></td>
</tr>
<tr>
<td>IsSubsystemVirtual</td>
<td>boolean — {'off'}</td>
<td>'on'</td>
</tr>
<tr>
<td></td>
<td>Read-only</td>
<td></td>
</tr>
<tr>
<td>Function-Call Generator</td>
<td>Sample time</td>
<td>{'1'}</td>
</tr>
<tr>
<td></td>
<td>Number of iterations</td>
<td>{'1'}</td>
</tr>
<tr>
<td>Function-Call Subsystem</td>
<td>Show port labels</td>
<td>'none'</td>
</tr>
<tr>
<td></td>
<td><strong>Note</strong> The values 'off' and 'on' are for backward compatibility only and should not be used in new models or when updating existing models.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Block choice</td>
<td>{''}</td>
</tr>
<tr>
<td></td>
<td>Template block</td>
<td>{''}</td>
</tr>
<tr>
<td>Block (Type)/Parameter</td>
<td>Dialog Box Prompt</td>
<td>Values</td>
</tr>
<tr>
<td>-------------------------------------</td>
<td>------------------------------------</td>
<td>-----------------------------------------------------------------------</td>
</tr>
<tr>
<td>MemberBlocks</td>
<td>Member blocks</td>
<td><code>{''}</code></td>
</tr>
<tr>
<td>Permissions</td>
<td>Read/Write permissions</td>
<td>{'ReadWrite'}</td>
</tr>
<tr>
<td>ErrorFcn</td>
<td>Name of error callback function</td>
<td><code>{''}</code></td>
</tr>
<tr>
<td>PermitHierarchical Resolution</td>
<td>Permit hierarchical resolution</td>
<td>{'All'}</td>
</tr>
<tr>
<td>TreatAsAtomicUnit</td>
<td>Treat as atomic unit</td>
<td>{'off'}</td>
</tr>
<tr>
<td>MinAlgLoopOccurrences</td>
<td>Minimize algebraic loop occurrences</td>
<td>{'off'}</td>
</tr>
<tr>
<td>PropExecContextOutsideSubsystem</td>
<td>Propagate execution context</td>
<td>{'off'}</td>
</tr>
<tr>
<td>CheckFcnCallInpInsideContextMsg</td>
<td>Warn if function-call inputs are</td>
<td>{'off'}</td>
</tr>
<tr>
<td>SystemSampleTime</td>
<td>Sample time (-1 for inherited)</td>
<td>{'-1'}</td>
</tr>
<tr>
<td>RTWSystemCode</td>
<td>Function packaging</td>
<td>{'Auto'}</td>
</tr>
<tr>
<td>RTWFcnNameOpts</td>
<td>Function name options</td>
<td>{'Auto'}</td>
</tr>
<tr>
<td>RTWFcnName</td>
<td>Function name</td>
<td><code>{''}</code></td>
</tr>
<tr>
<td>RTWFileNameOpts</td>
<td>File name options</td>
<td>{'Auto'}</td>
</tr>
<tr>
<td>RTWFileName</td>
<td>File name (no extension)</td>
<td><code>{''}</code></td>
</tr>
<tr>
<td>DataTypeOverride</td>
<td>No dialog box prompt</td>
<td>{'UseLocalSettings'}</td>
</tr>
</tbody>
</table>

Specifies data type used to override fixed-point data types. Set by **Data type override** on the Fixed-Point Tool.
<table>
<thead>
<tr>
<th>Block (Type)/Parameter</th>
<th>Dialog Box Prompt</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>MinMaxOverflowLogging</td>
<td>Setting for fixed-point instrumentation. Set by <strong>Fixed-point instrumentation mode</strong> on the Fixed-Point Tool.</td>
<td>{'UseLocalSettings'}</td>
</tr>
<tr>
<td>IsSubsystemVirtual</td>
<td></td>
<td>boolean — {'off'}</td>
</tr>
<tr>
<td>If(If)</td>
<td></td>
<td>Read-only</td>
</tr>
<tr>
<td>NumInputs</td>
<td>Number of inputs</td>
<td>{'1'}</td>
</tr>
<tr>
<td>IfExpression</td>
<td>If expression (e.g., u1 ~ = 0)</td>
<td>{'u1 &gt; 0'}</td>
</tr>
<tr>
<td>ElseIfExpressions</td>
<td>Elseif expressions (comma-separated list, e.g., u2 ~ = 0, u3(2) &lt; u2)</td>
<td>{}</td>
</tr>
<tr>
<td>ShowElse</td>
<td>Show else condition</td>
<td>'off'</td>
</tr>
<tr>
<td>ZeroCross</td>
<td>Enable zero-crossing detection</td>
<td>'off'</td>
</tr>
<tr>
<td>SampleTime</td>
<td>Sample time (-1 for inherited)</td>
<td>{'-1'}</td>
</tr>
<tr>
<td>If Action Subsystem (SubSystem)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ShowPortLabels</td>
<td>Show port labels</td>
<td>'none'</td>
</tr>
<tr>
<td>Note</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>The values 'off' and 'on' are for backward compatibility only and should not be used in new models or when updating existing models.</td>
<td></td>
</tr>
<tr>
<td>BlockChoice</td>
<td>Block choice</td>
<td>{''}</td>
</tr>
<tr>
<td>TemplateBlock</td>
<td>Template block</td>
<td>{''}</td>
</tr>
<tr>
<td>MemberBlocks</td>
<td>Member blocks</td>
<td>{''}</td>
</tr>
<tr>
<td>Permissions</td>
<td>Read/Write permissions</td>
<td>{'ReadWrite'}</td>
</tr>
</tbody>
</table>
### Block-Specific Parameters

<table>
<thead>
<tr>
<th>Block (Type)/Parameter</th>
<th>Dialog Box Prompt</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>ErrorFcn</td>
<td>Name of error callback function</td>
<td>{'''}</td>
</tr>
<tr>
<td>PermitHierarchical Resolution</td>
<td>Permit hierarchical resolution</td>
<td>{'All'}</td>
</tr>
<tr>
<td>TreatAsAtomicUnit</td>
<td>Treat as atomic unit</td>
<td>'off'</td>
</tr>
<tr>
<td>MinAlgLoopOccurrences</td>
<td>Minimize algebraic loop occurrences</td>
<td>{'off'}</td>
</tr>
<tr>
<td>PropExecContext</td>
<td>Propagate execution context across subsystem boundary</td>
<td>{'off'}</td>
</tr>
<tr>
<td>CheckFcnCallInp</td>
<td>Warn if function-call inputs are context-specific</td>
<td>{'off'}</td>
</tr>
<tr>
<td>SystemSampleTime</td>
<td>Sample time (-1 for inherited)</td>
<td>{'-1'}</td>
</tr>
<tr>
<td>RTWSYSTEMCode</td>
<td>Function packaging</td>
<td>{'Auto'}</td>
</tr>
<tr>
<td>RTWFcnNameOpts</td>
<td>Function name options</td>
<td>{'Auto'}</td>
</tr>
<tr>
<td>RTWFcnName</td>
<td>Function name</td>
<td>{'''}</td>
</tr>
<tr>
<td>RTWFileNameOpts</td>
<td>File name options</td>
<td>{'Auto'}</td>
</tr>
<tr>
<td>RTWFileName</td>
<td>File name (no extension)</td>
<td>{'''}</td>
</tr>
<tr>
<td>DataTypeOverride</td>
<td>No dialog box prompt</td>
<td>Specifies data type used to override fixed-point data types. Set by <strong>Data type override</strong> on the Fixed-Point Tool.</td>
</tr>
<tr>
<td>Block (Type)/Parameter</td>
<td>Dialog Box Prompt</td>
<td>Values</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>MinMaxOverflowLogging</td>
<td>Setting for fixed-point instrumentation. Set by <strong>Fixed-point instrumentation mode</strong> on the Fixed-Point Tool.</td>
<td>{'UseLocalSettings'}</td>
</tr>
<tr>
<td>IsSubsystemVirtual</td>
<td></td>
<td>boolean — {'off'}</td>
</tr>
<tr>
<td>In1 (Inport)</td>
<td></td>
<td>Read-only</td>
</tr>
<tr>
<td>Port</td>
<td>Port number</td>
<td>{'1'}</td>
</tr>
<tr>
<td>IconDisplay</td>
<td>Icon display</td>
<td>'Signal name'</td>
</tr>
<tr>
<td>LatchByDelayingOutsideSignal</td>
<td>Latch input by delaying outside signal</td>
<td>{'off'}</td>
</tr>
<tr>
<td>LatchInputForFeedbackSignals</td>
<td>Latch input for feedback signals of function-call subsystem outputs</td>
<td>{'off'}</td>
</tr>
<tr>
<td>Interpolate</td>
<td>Interpolate data</td>
<td>'off'</td>
</tr>
<tr>
<td>UseBusObject</td>
<td>Specify properties via bus object</td>
<td>{'off'}</td>
</tr>
<tr>
<td>BusObject</td>
<td>Bus object for specifying bus properties</td>
<td>{'BusObject'}</td>
</tr>
<tr>
<td>BusOutputAsStruct</td>
<td>Output as nonvirtual bus</td>
<td>{'off'}</td>
</tr>
<tr>
<td>PortDimensions</td>
<td>Port dimensions (-1 for inherited)</td>
<td>{'-1'}</td>
</tr>
<tr>
<td>SampleTime</td>
<td>Sample time (-1 for inherited)</td>
<td>{'-1'}</td>
</tr>
<tr>
<td>OutMin</td>
<td>Minimum</td>
<td>{'[]'}</td>
</tr>
<tr>
<td>OutMax</td>
<td>Maximum</td>
<td>{'[]'}</td>
</tr>
<tr>
<td>Block (Type)/Parameter</td>
<td>Dialog Box Prompt</td>
<td>Values</td>
</tr>
<tr>
<td>------------------------</td>
<td>-------------------</td>
<td>--------</td>
</tr>
<tr>
<td>OutDataTypeStr</td>
<td>Data type</td>
<td>{'Inherit: auto'}</td>
</tr>
<tr>
<td>LockScale</td>
<td>Lock output data type setting against changes by the fixed-point tools</td>
<td>{'off'}</td>
</tr>
<tr>
<td>Unit</td>
<td>Specify physical unit of the input signal to the block.</td>
<td>{'inherit'}</td>
</tr>
<tr>
<td>UnitNoProp</td>
<td>Specify physical unit of the input signal to the block without propagation. For a list of acceptable units, see Allowed Units.</td>
<td>'&lt;Enter unit&gt;'</td>
</tr>
<tr>
<td>SignalType</td>
<td>Signal type</td>
<td>{'auto'}</td>
</tr>
<tr>
<td>Model (ModelReference)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ModelNameDialog</td>
<td>The name of the referenced model exactly as you typed it in, with any surrounding whitespace removed. When you set ModelNameDialog programmatically or with the GUI, Simulink automatically sets the values of ModelName and ModelFile based on the value of ModelNameDialog.</td>
<td>{'&lt;Enter Model Name&gt;'}</td>
</tr>
<tr>
<td>Block (Type)/Parameter</td>
<td>Dialog Box Prompt</td>
<td>Values</td>
</tr>
<tr>
<td>------------------------</td>
<td>------------------</td>
<td>--------</td>
</tr>
<tr>
<td>ModelName</td>
<td>The value of ModelNameDialog stripped of any filename extension that you provided. For backward compatibility, setting ModelName programmatically actually sets ModelNameDialog, which then sets ModelName as described. You cannot use get_param to obtain the ModelName of a protected model, because the name without a suffix would be ambiguous. Use get_param on ModelFile instead. You can test ProtectedModel to determine programmatically whether a referenced model is protected.</td>
<td>character vector — Set automatically when ModelNameDialog is set.</td>
</tr>
<tr>
<td>ModelFile</td>
<td>The value of ModelNameDialog with a filename extension. The suffix of the first match Simulink finds becomes the suffix of ModelFile. Setting ModelFile programmatically actually sets ModelNameDialog, which then sets ModelFile as described.</td>
<td>character vector — Set automatically when ModelNameDialog is set.</td>
</tr>
<tr>
<td>ProtectedModel</td>
<td>Read-only boolean indicating whether the model referenced by the block is protected (on) or unprotected (off).</td>
<td>boolean — 'off'</td>
</tr>
<tr>
<td>InstanceParameters</td>
<td>Structure array of instance-specific parameters on the Model block.</td>
<td>structure with fields: Name, Value, Path, and Argument.</td>
</tr>
<tr>
<td>Block (Type)/Parameter</td>
<td>Dialog Box Prompt</td>
<td>Values</td>
</tr>
<tr>
<td>----------------------------------------</td>
<td>-----------------------------------------------------------------------------------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>ParameterArgumentNames</td>
<td>Names of instance-specific parameters that the referenced model defines.</td>
<td>{''}</td>
</tr>
<tr>
<td></td>
<td>Corresponds to the Name column in the table on the Instance parameters tab.</td>
<td></td>
</tr>
<tr>
<td>ParameterArgumentValues</td>
<td>Values for model arguments.</td>
<td>structure with no fields</td>
</tr>
<tr>
<td></td>
<td>Corresponds to the Value column in the table on the Instance parameters tab.</td>
<td></td>
</tr>
<tr>
<td>SimulationMode</td>
<td>Specifies whether to simulate the model by generating and executing code or by</td>
<td>{'Normal'}</td>
</tr>
<tr>
<td></td>
<td>interpreting the model in Simulink software.</td>
<td></td>
</tr>
<tr>
<td>Variant</td>
<td>Specifies whether the Model block references variant models or Variant Subsystems.</td>
<td>{'off'}</td>
</tr>
<tr>
<td>VariantConfigurationObject</td>
<td>Specifies the variant configuration object that is associated with the model.</td>
<td>{''}</td>
</tr>
<tr>
<td></td>
<td>The value is an empty character vector if no configuration object is associated;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>otherwise, it is the name of a Simulink.VariantConfigurationData object.</td>
<td></td>
</tr>
<tr>
<td>LabelModeActiveChoice</td>
<td>Specifies the selected choice when VariantControlMode is set to Label, and if so,</td>
<td>{''}</td>
</tr>
<tr>
<td></td>
<td>the name of that label mode active choice.</td>
<td>The value is the empty character vector if no label mode active choice is specified; or the name of the label mode active choice.</td>
</tr>
<tr>
<td>ActiveVariant</td>
<td>The variant that is currently active, either because its variant condition is true</td>
<td>{''}</td>
</tr>
<tr>
<td></td>
<td>or LabelModeActiveChoice has specified this variant.</td>
<td>The value is the empty character vector if no variant is active; or the name of the active variant.</td>
</tr>
</tbody>
</table>
### Block (Type)/Parameter

#### GeneratePreprocessor Conditionals
- **Dialog Box Prompt:** Locally controls whether generated code contains preprocessor conditionals. This parameter applies only to Simulink Coder code generation and has no effect on the behavior of a model in Simulink.
- **Values:**

  - {'off'} | 'on'

  The parameter is available only for ERT targets. For more information, see “Variant Systems” (Embedded Coder).

#### DefaultDataLogging
- **Values:**

  - {'off'} | 'on'

#### Out1 (Outport)

<table>
<thead>
<tr>
<th>Block Type/Parameter</th>
<th>Dialog Box Prompt</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Port</td>
<td>Port number</td>
<td>{'1'}</td>
</tr>
<tr>
<td>IconDisplay</td>
<td>Icon display</td>
<td>'Signal name'</td>
</tr>
<tr>
<td>UseBusObject</td>
<td>Specify properties via bus object</td>
<td>{'off'}</td>
</tr>
<tr>
<td>BusObject</td>
<td>Bus object for validating input bus</td>
<td>{'BusObject'}</td>
</tr>
<tr>
<td>BusOutputAsStruct</td>
<td>Output as nonvirtual bus in parent model</td>
<td>{'off'}</td>
</tr>
<tr>
<td>PortDimensions</td>
<td>Port dimensions (-1 for inherited)</td>
<td>{'-1'}</td>
</tr>
<tr>
<td>VarSizeSig</td>
<td>Variable-size signal</td>
<td>{'Inherit'}</td>
</tr>
<tr>
<td>SampleTime</td>
<td>Sample time (-1 for inherited)</td>
<td>{'-1'}</td>
</tr>
<tr>
<td>OutMin</td>
<td>Minimum</td>
<td>{'[]'}</td>
</tr>
<tr>
<td>OutMax</td>
<td>Maximum</td>
<td>{'[]'}</td>
</tr>
<tr>
<td>Block (Type)/Parameter</td>
<td>Dialog Box Prompt</td>
<td>Values</td>
</tr>
<tr>
<td>-----------------------</td>
<td>-------------------</td>
<td>--------</td>
</tr>
<tr>
<td>OutDataTypeStr</td>
<td>Data type</td>
<td>{'Inherit: auto'}</td>
</tr>
<tr>
<td>LockScale</td>
<td>Lock output data type setting against changes by the fixed-point tools</td>
<td>{'off'}</td>
</tr>
<tr>
<td>Unit</td>
<td>Specify physical unit of the input signal to the block. For a list of acceptable units, see Allowed Units.</td>
<td>{'inherit'}</td>
</tr>
<tr>
<td>UnitNoProp</td>
<td>Specify physical unit of the input signal to the block without propagation. For a list of acceptable units, see Allowed Units.</td>
<td>'&lt;Enter unit&gt;'</td>
</tr>
<tr>
<td>SignalObject</td>
<td>This parameter does not appear in the block dialog box. Use the Model Data Editor instead. See “Design Data Interface by Configuring Inport and Outport Blocks” (Simulink Coder).</td>
<td>Simulink.Signal object</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Object of a class that is derived from Simulink.Signal</td>
</tr>
<tr>
<td>StorageClass</td>
<td>This parameter does not appear in the block dialog box. Use the Model Data Editor instead. See “Design Data Interface by Configuring Inport and Outport Blocks” (Simulink Coder).</td>
<td>{'Auto'}</td>
</tr>
<tr>
<td>SignalName</td>
<td>Signal name</td>
<td>character vector</td>
</tr>
<tr>
<td>Block (Type)/Parameter</td>
<td>Dialog Box Prompt</td>
<td>Values</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>---------------------</td>
<td>--------------------------------------------</td>
</tr>
<tr>
<td>SignalType</td>
<td>Signal type</td>
<td>{'auto'}</td>
</tr>
<tr>
<td>OutputWhenDisabled</td>
<td>Output when disabled</td>
<td>{'held'}</td>
</tr>
<tr>
<td>InitialOutput</td>
<td>Initial output</td>
<td>{'[]'}</td>
</tr>
<tr>
<td>MustResolveToSignalObject</td>
<td>This parameter does not appear in the block dialog box. Use the Model Data Editor instead. See “For Signals”.</td>
<td>{'off'}</td>
</tr>
<tr>
<td>Subsystem (SubSystem)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ShowPortLabels</td>
<td>Show port labels</td>
<td>'none'</td>
</tr>
<tr>
<td>Note</td>
<td>The values 'off' and 'on' are for backward compatibility only and should not be used in new models or when updating existing models.</td>
<td></td>
</tr>
<tr>
<td>BlockChoice</td>
<td>Block choice</td>
<td>{''}</td>
</tr>
<tr>
<td>TemplateBlock</td>
<td>Template block</td>
<td>{''}</td>
</tr>
<tr>
<td>MemberBlocks</td>
<td>Member blocks</td>
<td>{''}</td>
</tr>
<tr>
<td>Permissions</td>
<td>Read/Write permissions</td>
<td>{'ReadWrite'}</td>
</tr>
<tr>
<td>ErrorFcn</td>
<td>Name of error callback function</td>
<td>{''}</td>
</tr>
<tr>
<td>PermitHierarchicalResolution</td>
<td>Permit hierarchical resolution</td>
<td>{'All'}</td>
</tr>
<tr>
<td>TreatAsAtomicUnit</td>
<td>Treat as atomic unit</td>
<td>{'off'}</td>
</tr>
<tr>
<td>TreatAsGroupedWhenPropagatingVariantConditions</td>
<td>Treat as grouped when propagating variant conditions</td>
<td>'off'</td>
</tr>
<tr>
<td>VariantControl</td>
<td>Variant control</td>
<td>{'Variant'}</td>
</tr>
<tr>
<td>MinAlgLoopOccurrences</td>
<td>Minimize algebraic loop occurrences</td>
<td>{'off'}</td>
</tr>
<tr>
<td>Block (Type)/Parameter</td>
<td>Dialog Box Prompt</td>
<td>Values</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>---------------------------------------------------------------------------------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>PropExecContext OutsideSubsystem</td>
<td>Propagate execution context across subsystem boundary</td>
<td>{'off'}</td>
</tr>
<tr>
<td>CheckFcnCallInp InsideContextMsg</td>
<td>Warn if function-call inputs are context-specific</td>
<td>{'off'}</td>
</tr>
<tr>
<td>SystemSampleTime</td>
<td>Sample time (-1 for inherited)</td>
<td>{'-1'}</td>
</tr>
<tr>
<td>RTWSystemCode</td>
<td>Code generation function packaging</td>
<td>{'Auto'}</td>
</tr>
<tr>
<td>RTWFcnNameOpts</td>
<td>Code generation function name options</td>
<td>{'Auto'}</td>
</tr>
<tr>
<td>RTWFcnName</td>
<td>Code generation function name</td>
<td>{''}</td>
</tr>
<tr>
<td>RTWFileNameOpts</td>
<td>Code generation file name options</td>
<td>{'Auto'}</td>
</tr>
<tr>
<td>RTWFileName</td>
<td>Code generation file name (no extension)</td>
<td>{''}</td>
</tr>
<tr>
<td>DataTypeOverride</td>
<td>Specifies data type used to override fixed-point data types.</td>
<td>{'UseLocalSettings'}</td>
</tr>
<tr>
<td>MinMaxOverflowLogging</td>
<td>No dialog box prompt</td>
<td>{'UseLocalSettings'}</td>
</tr>
<tr>
<td>IsSubsystemVirtual</td>
<td>boolean — {'on'}</td>
<td>'off'</td>
</tr>
<tr>
<td>Virtual</td>
<td>For internal use</td>
<td></td>
</tr>
<tr>
<td>Switch Case (SwitchCase)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Block (Type)/Parameter</td>
<td>Dialog Box Prompt</td>
<td>Values</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>----------------------------------------</td>
<td>-----------------------------------</td>
</tr>
<tr>
<td>CaseConditions</td>
<td>Case conditions (e.g., {1,2,3})</td>
<td>{'1}'</td>
</tr>
<tr>
<td>ShowDefaultCase</td>
<td>Show default case</td>
<td>‘off’</td>
</tr>
<tr>
<td>ZeroCross</td>
<td>Enable zero-crossing detection</td>
<td>‘off’</td>
</tr>
<tr>
<td>SampleTime</td>
<td>Sample time (-1 for inherited)</td>
<td>{'-1}'</td>
</tr>
<tr>
<td>CaseShowDefault</td>
<td>Deprecated in R2009b</td>
<td></td>
</tr>
<tr>
<td>Switch Case Action Subsystem</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ShowPortLabels</td>
<td>Show port labels</td>
<td>‘none’</td>
</tr>
<tr>
<td>BlockChoice</td>
<td>Block choice</td>
<td>{}</td>
</tr>
<tr>
<td>TemplateBlock</td>
<td>Template block</td>
<td>{}</td>
</tr>
<tr>
<td>MemberBlocks</td>
<td>Member blocks</td>
<td>{}</td>
</tr>
<tr>
<td>Permissions</td>
<td>Read/Write permissions</td>
<td>{'ReadWrite}'</td>
</tr>
<tr>
<td>ErrorFcn</td>
<td>Name of error callback function</td>
<td>{}</td>
</tr>
<tr>
<td>PermitHierarchical Resolution</td>
<td>Permit hierarchical resolution</td>
<td>{'All}'</td>
</tr>
<tr>
<td>TreatAsAtomicUnit</td>
<td>Treat as atomic unit</td>
<td>‘off’</td>
</tr>
<tr>
<td>MinAlgLoopOccurrences</td>
<td>Minimize algebraic loop occurrences</td>
<td>‘off’</td>
</tr>
<tr>
<td>PropExecContextOutsideSubsystem</td>
<td>Propagate execution context across subsystem boundary</td>
<td>‘off’</td>
</tr>
<tr>
<td>CheckFcnCallInpInsideContextMsg</td>
<td>Warn if function-call inputs are context-specific</td>
<td>‘off’</td>
</tr>
<tr>
<td>SystemSampleTime</td>
<td>Sample time (-1 for inherited)</td>
<td>{'-1}'</td>
</tr>
<tr>
<td>Block (Type)/Parameter</td>
<td>Dialog Box Prompt</td>
<td>Values</td>
</tr>
<tr>
<td>----------------------------</td>
<td>-----------------------------------------------</td>
<td>------------------------------------------------------------------------</td>
</tr>
<tr>
<td>RTWSystemCode</td>
<td>Code generation function packaging</td>
<td>{'Auto'}</td>
</tr>
<tr>
<td>RTWFcnNameOpts</td>
<td>Code generation function name options</td>
<td>{'Auto'}</td>
</tr>
<tr>
<td>RTWFcnName</td>
<td>Code generation function name</td>
<td>{}</td>
</tr>
<tr>
<td>RTWFileNameOpts</td>
<td>Code generation file name options</td>
<td>{'Auto'}</td>
</tr>
<tr>
<td>RTWFileName</td>
<td>Code generation file name (no extension)</td>
<td>{}</td>
</tr>
<tr>
<td>DataTypeOverride</td>
<td>No dialog box prompt</td>
<td>{'UseLocalSettings'}</td>
</tr>
<tr>
<td>MinMaxOverflowLogging</td>
<td>No dialog box prompt</td>
<td>{'UseLocalSettings'}</td>
</tr>
<tr>
<td>IsSubsystemVirtual</td>
<td>boolean — {'off'}</td>
<td>'on'</td>
</tr>
<tr>
<td>Trigger(TriggerPort)</td>
<td>Trigger type</td>
<td>{'rising'}</td>
</tr>
<tr>
<td>IsSimulinkFunction</td>
<td>Configure the Function-call subsystem to be a Simulink Function</td>
<td>{'off'}</td>
</tr>
<tr>
<td>Block (Type)/Parameter</td>
<td>Dialog Box Prompt</td>
<td>Values</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-------------------------------------------------------------</td>
<td>---------------------------------------------</td>
</tr>
<tr>
<td>StatesWhenEnabling</td>
<td>States when enabling</td>
<td>{'held'}</td>
</tr>
<tr>
<td>PropagateVarSize</td>
<td>Propagate sizes of variable-size signals</td>
<td>{'During execution'}</td>
</tr>
<tr>
<td>ShowOutputPort</td>
<td>Show output port</td>
<td>{'off'}</td>
</tr>
<tr>
<td>OutputDataType</td>
<td>Output data type</td>
<td>{'auto'}</td>
</tr>
<tr>
<td>SampleTimeType</td>
<td>Sample time type</td>
<td>{'triggered'}</td>
</tr>
<tr>
<td>SampleTime</td>
<td>Sample time</td>
<td>{'1'}</td>
</tr>
<tr>
<td>ZeroCross</td>
<td>Enable zero-crossing detection</td>
<td>'off'</td>
</tr>
<tr>
<td>PortDimensions</td>
<td>Port dimensions (-1 for inherited)</td>
<td>{'-1'}</td>
</tr>
<tr>
<td>TriggerSignalSampleTime</td>
<td>Trigger signal sample time</td>
<td>{'-1'}</td>
</tr>
<tr>
<td>OutMin</td>
<td>Minimum</td>
<td>{'[]'}</td>
</tr>
<tr>
<td>OutMax</td>
<td>Maximum</td>
<td>{'[]'}</td>
</tr>
<tr>
<td>OutDataTypeStr</td>
<td>Data type</td>
<td>{'Inherit: auto'}</td>
</tr>
<tr>
<td>Interpolate</td>
<td>Interpolate data</td>
<td>'off'</td>
</tr>
</tbody>
</table>

Triggered Subsystem (SubSystem)
<table>
<thead>
<tr>
<th>Block (Type)/Parameter</th>
<th>Dialog Box Prompt</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>ShowPortLabels</td>
<td>Show port labels</td>
<td>'none'</td>
</tr>
<tr>
<td><strong>Note</strong> The values 'off' and 'on' are for backward compatibility only and should not be used in new models or when updating existing models.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BlockChoice</td>
<td>Block choice</td>
<td>{''}</td>
</tr>
<tr>
<td>TemplateBlock</td>
<td>Template block</td>
<td>{''}</td>
</tr>
<tr>
<td>MemberBlocks</td>
<td>Member blocks</td>
<td>{''}</td>
</tr>
<tr>
<td>Permissions</td>
<td>Read/Write permissions</td>
<td>{'ReadWrite'}</td>
</tr>
<tr>
<td>ErrorFcn</td>
<td>Name of error callback function</td>
<td>{''}</td>
</tr>
<tr>
<td>PermitHierarchicalResolution</td>
<td>Permit hierarchical resolution</td>
<td>{'All'}</td>
</tr>
<tr>
<td>TreatAsAtomicUnit</td>
<td>Treat as atomic unit</td>
<td>'off'</td>
</tr>
<tr>
<td>MinAlgLoopOccurrences</td>
<td>Minimize algebraic loop occurrences</td>
<td>{'off'}</td>
</tr>
<tr>
<td>PropExecContext</td>
<td>Propagate execution context across subsystem boundary</td>
<td>{'off'}</td>
</tr>
<tr>
<td>OutsideSubsystem</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CheckFcnCallInp</td>
<td>Warn if function-call inputs are context-specific</td>
<td>{'off'}</td>
</tr>
<tr>
<td>InsideContextMsg</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SystemSampleTime</td>
<td>Sample time (-1 for inherited)</td>
<td>{'-1'}</td>
</tr>
<tr>
<td>RTWSystemCode</td>
<td>Code generation function packaging</td>
<td>{'Auto'}</td>
</tr>
<tr>
<td>RTWFcnNameOpts</td>
<td>Code generation function name options</td>
<td>{'Auto'}</td>
</tr>
<tr>
<td>RTWFcnName</td>
<td>Code generation function name</td>
<td>{''}</td>
</tr>
<tr>
<td>Block (Type)/Parameter</td>
<td>Dialog Box Prompt</td>
<td>Values</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>----------------------------------------------------------------------------------</td>
<td>--------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>RTWFileNameOpts</td>
<td>Code generation file name options</td>
<td>{'Auto'}</td>
</tr>
<tr>
<td>RTWFileName</td>
<td>Code generation file name (no extension)</td>
<td>{''}</td>
</tr>
<tr>
<td>DataTypeOverride</td>
<td>No dialog box prompt</td>
<td>{'UseLocalSettings'}</td>
</tr>
<tr>
<td>MinMaxOverflowLogging</td>
<td>No dialog box prompt</td>
<td>{'UseLocalSettings'}</td>
</tr>
<tr>
<td>IsSubsystemVirtual</td>
<td></td>
<td>boolean — {'off'}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Read-only</td>
</tr>
<tr>
<td>Unit Conversion</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OutDataTypeStr</td>
<td>Output data type</td>
<td>{'Inherit via internal rule'}</td>
</tr>
<tr>
<td>Unit System Configuration</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AllowAllUnitSystems</td>
<td>Allow or restrict unit systems.</td>
<td>boolean — {'on'}</td>
</tr>
<tr>
<td>UnitSystems</td>
<td>Displays allowed unit system.</td>
<td>cell array of character vectors — {'SI','English',SI (extended),'CGS'}</td>
</tr>
<tr>
<td>While Iterator (WhileIterator)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MaxIters</td>
<td>Maximum number of iterations (-1 for unlimited)</td>
<td>{'5'}</td>
</tr>
<tr>
<td>WhileBlockType</td>
<td>While loop type</td>
<td>{'while'}</td>
</tr>
</tbody>
</table>

6-238
<table>
<thead>
<tr>
<th>Block (Type)/Parameter</th>
<th>Dialog Box Prompt</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reset States</td>
<td>States when starting</td>
<td>{'held'}</td>
</tr>
<tr>
<td>Show Iteration Port</td>
<td>Show iteration number port</td>
<td>{'off'}</td>
</tr>
<tr>
<td>Output Data Type</td>
<td>Output data type</td>
<td>{'int32'}</td>
</tr>
<tr>
<td>While Iterator Subsystem (SubSystem)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Show Port Labels</td>
<td>Show port labels</td>
<td>'none'</td>
</tr>
<tr>
<td>Note</td>
<td>The values 'off' and 'on' are for backward compatibility only and should not be used in new models or when updating existing models.</td>
<td></td>
</tr>
<tr>
<td>Block Choice</td>
<td>Block choice</td>
<td>{''}</td>
</tr>
<tr>
<td>Template Block</td>
<td>Template block</td>
<td>{''}</td>
</tr>
<tr>
<td>Member Blocks</td>
<td>Member blocks</td>
<td>{''}</td>
</tr>
<tr>
<td>Permissions</td>
<td>Read/Write permissions</td>
<td>{'ReadWrite'}</td>
</tr>
<tr>
<td>ErrorFcn</td>
<td>Name of error callback function</td>
<td>{''}</td>
</tr>
<tr>
<td>Permit Hierarchical Resolution</td>
<td>Permit hierarchical resolution</td>
<td>{'All'}</td>
</tr>
<tr>
<td>Treat As Atomic Unit</td>
<td>Treat as atomic unit</td>
<td>'off'</td>
</tr>
<tr>
<td>Min Alg Loop Occurrences</td>
<td>Minimize algebraic loop occurrences</td>
<td>{'off'}</td>
</tr>
<tr>
<td>Prop Execute Context</td>
<td>Propagate execution context across subsystem boundary</td>
<td>{'off'}</td>
</tr>
<tr>
<td>CheckFcn Call Inp</td>
<td>Warn if function-call inputs are context-specific</td>
<td>{'off'}</td>
</tr>
<tr>
<td>System Sample Time</td>
<td>Sample time (-1 for inherited)</td>
<td>{'-1'}</td>
</tr>
<tr>
<td>Block (Type)/Parameter</td>
<td>Dialog Box Prompt</td>
<td>Values</td>
</tr>
<tr>
<td>------------------------------</td>
<td>--------------------------------------------------------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>RTWSystemCode</td>
<td>Code generation function packaging</td>
<td>{'Auto'}</td>
</tr>
<tr>
<td>RTWFcnNameOpts</td>
<td>Code generation function name options</td>
<td>{'Auto'}</td>
</tr>
<tr>
<td>RTWFcnName</td>
<td>Code generation function name</td>
<td>{}</td>
</tr>
<tr>
<td>RTWFileNameOpts</td>
<td>Code generation file name options</td>
<td>{'Auto'}</td>
</tr>
<tr>
<td>RTWFileName</td>
<td>Code generation file name (no extension)</td>
<td>{}</td>
</tr>
<tr>
<td>DataTypeOverride</td>
<td>No dialog box prompt</td>
<td>{'UseLocalSettings'}</td>
</tr>
<tr>
<td>MinMaxOverflowLogging</td>
<td>No dialog box prompt</td>
<td>{'UseLocalSettings'}</td>
</tr>
<tr>
<td>IsSubsystemVirtual</td>
<td>boolean — {'off'}</td>
<td>'on'</td>
</tr>
</tbody>
</table>

Model and Block Parameters

6-240
## Signal Attributes Library Block Parameters

<table>
<thead>
<tr>
<th>Block (Type)/Parameter</th>
<th>Dialog Box Prompt</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Bus to Vector (BusToVector)</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data Type Conversion (DataTypeConversion)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OutMin</td>
<td>Output minimum</td>
<td>{'[]}</td>
</tr>
<tr>
<td>OutMax</td>
<td>Output maximum</td>
<td>{'[]}</td>
</tr>
<tr>
<td>OutDataTypeStr</td>
<td>Output data type</td>
<td>{'Inherit: Inherit via back propagation'}</td>
</tr>
<tr>
<td>LockScale</td>
<td>Lock output data type setting against changes by the fixed-point tools</td>
<td>{'off'}</td>
</tr>
<tr>
<td>ConvertRealWorld</td>
<td>Input and output to have equal</td>
<td>{'Real World Value (RWV)'}</td>
</tr>
<tr>
<td>RndMeth</td>
<td>Integer rounding mode</td>
<td>'Ceiling'</td>
</tr>
<tr>
<td>SaturateOnInteger Overflow</td>
<td>Saturate on integer overflow</td>
<td>{'off'}</td>
</tr>
<tr>
<td>SampleTime</td>
<td>Sample time (-1 for inherited)</td>
<td>{'-1'}</td>
</tr>
<tr>
<td><strong>Data Type Conversion Inherited (Conversion Inherited) (masked subsystem)</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ConvertRealWorld</td>
<td>Input and Output to have equal</td>
<td>{'Real World Value'}</td>
</tr>
</tbody>
</table>
### Block (Type)/Parameter

<table>
<thead>
<tr>
<th>Block (Type)/Parameter</th>
<th>Dialog Box Prompt</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>RndMeth</strong></td>
<td>Integer rounding mode</td>
<td>'Ceiling'</td>
</tr>
<tr>
<td><strong>DoSatur</strong></td>
<td>Saturate to max or min when overflows occur</td>
<td>{'off'}</td>
</tr>
</tbody>
</table>

### Data Type Duplicate (DataTypeDuplicate)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Dialog Box Prompt</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>NumInputPorts</strong></td>
<td>Number of input ports</td>
<td>{'2'}</td>
</tr>
</tbody>
</table>

### Data Type Propagation (Data Type Propagation) (masked subsystem)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Dialog Box Prompt</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PropDataTypeMode</strong></td>
<td>1. Propagated data type</td>
<td>'Specify via dialog'</td>
</tr>
<tr>
<td><strong>PropDataType</strong></td>
<td>1.1. Propagated data type (e.g., fixdt(1,16), fixdt('single'))</td>
<td>{'fixdt(1,16)'}</td>
</tr>
<tr>
<td><strong>IfRefDouble</strong></td>
<td>1.1. If any reference input is double, output is</td>
<td>{'double'}</td>
</tr>
<tr>
<td><strong>IfRefSingle</strong></td>
<td>1.2. If any reference input is single, output is</td>
<td>'double'</td>
</tr>
<tr>
<td><strong>IsSigned</strong></td>
<td>1.3. Is-Signed</td>
<td>'IsSigned1'</td>
</tr>
<tr>
<td><strong>NumBitsBase</strong></td>
<td>1.4.1. Number-of-Bits: Base</td>
<td>'NumBits1'</td>
</tr>
<tr>
<td><strong>NumBitsMult</strong></td>
<td>1.4.2. Number-of-Bits: Multiplicative adjustment</td>
<td>{'1'}</td>
</tr>
<tr>
<td>Block (Type)/Parameter</td>
<td>Dialog Box Prompt</td>
<td>Values</td>
</tr>
<tr>
<td>-----------------------</td>
<td>-------------------</td>
<td>--------</td>
</tr>
<tr>
<td>NumBitsAdd</td>
<td>1.4.3. Number-of-Bits: Additive adjustment</td>
<td>{'0'}</td>
</tr>
<tr>
<td>NumBitsAllowFinal</td>
<td>1.4.4. Number-of-Bits: Allowable final values</td>
<td>{'1:128'}</td>
</tr>
<tr>
<td>PropScalingMode</td>
<td>2. Propagated scaling</td>
<td>'Specify via dialog'</td>
</tr>
<tr>
<td>PropScaling</td>
<td>2.1. Propagated scaling: Slope or [Slope Bias] ex. 2^-9</td>
<td>{'2^-10'}</td>
</tr>
<tr>
<td>ValuesUsedBestPrec</td>
<td>2.1. Values used to determine best precision scaling</td>
<td>{'[5 -7]'}</td>
</tr>
<tr>
<td>SlopeBase</td>
<td>2.1.1. Slope: Base</td>
<td>'Slope1'</td>
</tr>
<tr>
<td>SlopeMult</td>
<td>2.1.2. Slope: Multiplicative adjustment</td>
<td>{'1'}</td>
</tr>
<tr>
<td>SlopeAdd</td>
<td>2.1.3. Slope: Additive adjustment</td>
<td>{'0'}</td>
</tr>
</tbody>
</table>
## Block (Type)/Parameter

<table>
<thead>
<tr>
<th>Block (Type)/Parameter</th>
<th>Dialog Box Prompt</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>BiasBase</td>
<td>2.2.1. Bias: Base</td>
<td>{'Bias1}'</td>
</tr>
<tr>
<td>BiasMult</td>
<td>2.2.2. Bias: Multiplicative adjustment</td>
<td>{'1}'</td>
</tr>
<tr>
<td>BiasAdd</td>
<td>2.2.3. Bias: Additive adjustment</td>
<td>{'0}'</td>
</tr>
</tbody>
</table>

### Data Type Scaling Strip (ScalingStrip) (masked subsystem)

#### IC (InitialCondition)

<table>
<thead>
<tr>
<th>Value</th>
<th>Initial value</th>
<th>{'1}'</th>
</tr>
</thead>
</table>

#### Probe (Probe)

| ProbeWidth | Probe width | \'off\' | \{'on\}' |
|------------|-------------|---------|
| ProbeSampleTime | Probe sample time | \'off\' | \{'on\}' |
| ProbeComplexSignal | Detect complex signal | \'off\' | \{'on\}' |
| ProbeSignalDimensions | Probe signal dimensions | \'off\' | \{'on\}' |
| ProbeFramedSignal | Detect framed signal | \'off\' | \{'on\}' |

<table>
<thead>
<tr>
<th>ProbeWidthDataType</th>
<th>Data type for width</th>
<th>{'double}'</th>
<th>{'single}'</th>
<th>'int8}'</th>
<th>'uint8}'</th>
<th>'int16}'</th>
<th>'uint16}'</th>
<th>'int32}'</th>
<th>'uint32}'</th>
<th>'Same as input}'</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>ProbeSampleTimeDataType</th>
<th>Data type for sample time</th>
<th>{'double}'</th>
<th>{'single}'</th>
<th>'int8}'</th>
<th>'uint8}'</th>
<th>'int16}'</th>
<th>'uint16}'</th>
<th>'int32}'</th>
<th>'uint32}'</th>
<th>'Same as input}'</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block (Type)/Parameter</td>
<td>Dialog Box Prompt</td>
<td>Values</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-------------------------------</td>
<td>----------------------------------------</td>
<td>------------------------------------------------------------------------</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ProbeComplexityDataType</td>
<td>Data type for signal complexity</td>
<td>{'double'}, 'single', 'int8', 'uint8', 'int16', 'uint16', 'int32', 'uint32', 'boolean', 'Same as input'</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ProbeDimensionsDataType</td>
<td>Data type for signal dimensions</td>
<td>{'double'}, 'single', 'int8', 'uint8', 'int16', 'uint16', 'int32', 'uint32', 'Same as input'</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ProbeFrameDataType</td>
<td>Data type for signal frames</td>
<td>{'double'}, 'single', 'int8', 'uint8', 'int16', 'uint16', 'int32', 'uint32', 'boolean', 'Same as input'</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Rate Transition (RateTransition)**

<table>
<thead>
<tr>
<th>Block (Type)/Parameter</th>
<th>Dialog Box Prompt</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integrity</td>
<td>Ensure data integrity during data transfer</td>
<td>'off', {'on'}</td>
</tr>
<tr>
<td>Deterministic</td>
<td>Ensure deterministic data transfer (maximum delay)</td>
<td>'off', {'on'}</td>
</tr>
<tr>
<td>InitialCondition</td>
<td>Initial conditions</td>
<td>{'0'}</td>
</tr>
<tr>
<td>OutPortSampleTimeOpt</td>
<td>Output port sample time options</td>
<td>{'Specify'}, 'Inherit', 'Multiple of input port sample time'</td>
</tr>
<tr>
<td>OutPortSampleTimeMultiple</td>
<td>Sample time multiple (&gt;0)</td>
<td>{'1'}</td>
</tr>
<tr>
<td>OutPortSampleTime</td>
<td>Output port sample time</td>
<td>{'-1'}</td>
</tr>
</tbody>
</table>

**Signal Conversion (SignalConversion)**

<table>
<thead>
<tr>
<th>Block (Type)/Parameter</th>
<th>Dialog Box Prompt</th>
<th>Values</th>
</tr>
</thead>
</table>
| ConversionOutput              | Output            | {'Signal copy'}, 'Virtual bus', 'Nonvirtual bus'}
<table>
<thead>
<tr>
<th>Block (Type)/Parameter</th>
<th>Dialog Box Prompt</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>OutDataTypeStr</td>
<td>Data type</td>
<td>{'Inherit: auto'}</td>
</tr>
<tr>
<td>OverrideOpt</td>
<td>Exclude this block from 'Block reduction' optimization</td>
<td>{'off'}</td>
</tr>
</tbody>
</table>

**Signal Specification (SignalSpecification)**

<table>
<thead>
<tr>
<th>Block (Type)/Parameter</th>
<th>Dialog Box Prompt</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dimensions</td>
<td>Dimensions (-1 for inherited)</td>
<td>{'-1'}</td>
</tr>
<tr>
<td>VarSizeSig</td>
<td>Variable-size signal</td>
<td>{'Inherit'}</td>
</tr>
<tr>
<td>SampleTime</td>
<td>Sample time (-1 for inherited)</td>
<td>{'-1'}</td>
</tr>
<tr>
<td>OutMin</td>
<td>Minimum</td>
<td>{'[]}'}</td>
</tr>
<tr>
<td>OutMax</td>
<td>Maximum</td>
<td>{'[]}'}</td>
</tr>
<tr>
<td>OutDataTypeStr</td>
<td>Data type</td>
<td>{'Inherit: auto'}</td>
</tr>
<tr>
<td>BusOutputAsStruct</td>
<td>Require nonvirtual bus</td>
<td>{'off'}</td>
</tr>
<tr>
<td>Unit</td>
<td>Specify physical unit of the input signal to the block. For a list of acceptable units, see Allowed Units.</td>
<td>{'inherit'}</td>
</tr>
<tr>
<td>UnitNoProp</td>
<td>Specify physical unit of the input signal to the block without propagation. For a list of acceptable units, see Allowed Units.</td>
<td>'&lt;Enter unit&gt;'</td>
</tr>
<tr>
<td>Block (Type)/Parameter</td>
<td>Dialog Box Prompt</td>
<td>Values</td>
</tr>
<tr>
<td>------------------------</td>
<td>-----------------------------------------------------------------------------------</td>
<td>---------------------------------------------</td>
</tr>
<tr>
<td>LockScale</td>
<td>Lock output data type setting against changes by the fixed-point tools</td>
<td>{'off'}</td>
</tr>
<tr>
<td>SignalType</td>
<td>Signal type</td>
<td>{'auto'}</td>
</tr>
<tr>
<td>Weighted Sample Time (SampleTimeMath)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TsampMathOp</td>
<td>Operation</td>
<td>'+'</td>
</tr>
<tr>
<td>weightValue</td>
<td>Weight value</td>
<td>{'1.0'}</td>
</tr>
<tr>
<td>TsampMathImp</td>
<td>Implement using</td>
<td>{'Online Calculations'}</td>
</tr>
<tr>
<td>OutDataTypeStr</td>
<td>Output data type</td>
<td>{'Inherit via internal rule'}</td>
</tr>
<tr>
<td>RndMeth</td>
<td>Integer rounding mode</td>
<td>'Ceiling'</td>
</tr>
<tr>
<td>SaturateOnInteger Overflow</td>
<td>Saturate on integer overflow</td>
<td>{'off'}</td>
</tr>
<tr>
<td>OutputDataTypeScaling Mode</td>
<td>Deprecated in R2009b</td>
<td></td>
</tr>
<tr>
<td>DoSatur</td>
<td>Deprecated in R2009b</td>
<td></td>
</tr>
<tr>
<td>Width (Width)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OutputDataTypeScaling Mode</td>
<td>Output data type mode</td>
<td>{'Choose intrinsic data type'}</td>
</tr>
<tr>
<td>Block (Type)/Parameter</td>
<td>Dialog Box Prompt</td>
<td>Values</td>
</tr>
<tr>
<td>-----------------------</td>
<td>--------------------</td>
<td>---------------------------------------------</td>
</tr>
<tr>
<td></td>
<td>Output data type</td>
<td>{'double'}, 'single', 'int8', 'uint8', 'int16', 'uint16', 'int32', 'uint32'</td>
</tr>
</tbody>
</table>
 Signal Routing Library Block Parameters

<table>
<thead>
<tr>
<th>Block (Type)/Parameter</th>
<th>Dialog Box Prompt</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Bus Assignment</strong> <strong>(BusAssignment)</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AssignedSignals</td>
<td>Signals that are being assigned</td>
<td>{''}</td>
</tr>
<tr>
<td>InputSignals</td>
<td>Signals in the bus</td>
<td>matrix — {'{}'}</td>
</tr>
<tr>
<td><strong>Bus Creator</strong> <strong>(BusCreator)</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
| InheritFromInputs            | Override bus signal names from inputs | {'on'} | 'off'  
If set to 'on', overrides bus signal names from inputs. Otherwise, inherits bus signal names from a bus object. |
| Inputs                       | Number of inputs                   | {'2'}                                                               |
| DisplayOption                |                                     | 'none' | 'signals' | {'bar'}  |
| NonVirtualBus                | Output as nonvirtual bus           | {'off'} | 'on'     |
| OutDataTypeStr               | Data type                           | {'Inherit: auto'} | 'double' | 'single' | 'int8' | 'uint8' | 'int16' | 'uint16' | 'int32' | 'uint32' | 'int64' | 'uint64' | 'boolean' | 'fixdt(1,16,0)' | 'fixdt(1,16,2^0,0)' | 'Enum: <class name>' | 'Bus: <object name>' |
| **Bus Selector** **(BusSelector)** |                                    |                                                                     |
| OutputSignals                | Selected signals                    | character vector — in the form 'signal1,signal2'                    |
| OutputAsBus                  | Output as bus                       | {'off'} | 'on'     |
| InputSignals                 | Signals in bus                      | matrix — {'{}'}                                                   |

Data Store Memory **(DataStoreMemory)**
<table>
<thead>
<tr>
<th>Block (Type)/Parameter</th>
<th>Dialog Box Prompt</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>DataStoreName</td>
<td>Data store name</td>
<td>{'A'}</td>
</tr>
<tr>
<td>ReadBeforeWriteMsg</td>
<td>Detect read before write</td>
<td>'none'</td>
</tr>
<tr>
<td>WriteAfterWriteMsg</td>
<td>Detect write after write</td>
<td>'none'</td>
</tr>
<tr>
<td>WriteAfterReadMsg</td>
<td>Detect write after read</td>
<td>'none'</td>
</tr>
<tr>
<td>InitialValue</td>
<td>Initial value</td>
<td>{'0'}</td>
</tr>
<tr>
<td>StateMustResolveTo</td>
<td>Data store name must resolve to Simulink signal object</td>
<td>{'off'}</td>
</tr>
<tr>
<td>SignalObject</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DataLogging</td>
<td>Log Signal Data</td>
<td>'off'</td>
</tr>
<tr>
<td>DataLoggingNameMode</td>
<td>Logging Name</td>
<td>{'SignalName'}</td>
</tr>
<tr>
<td>DataLoggingName</td>
<td>Logging Name</td>
<td>{''}</td>
</tr>
<tr>
<td>DataLoggingLimitPoints</td>
<td>Limit data points to last</td>
<td>'off'</td>
</tr>
<tr>
<td>DataLoggingMaxPoints</td>
<td>Limit data points to last</td>
<td>non-zero integer</td>
</tr>
<tr>
<td>DataLoggingDecimateData</td>
<td>Decimation</td>
<td>'off'</td>
</tr>
<tr>
<td>DataLoggingLimitPoints</td>
<td>Decimation</td>
<td>non-zero integer</td>
</tr>
<tr>
<td>StateStorageClass</td>
<td>Storage class</td>
<td>{'Auto'}</td>
</tr>
<tr>
<td>StateSignalObject</td>
<td>Signal object class</td>
<td>Simulink.Signal object</td>
</tr>
<tr>
<td></td>
<td>Storage class</td>
<td>Object of a class that is derived from Simulink.Signal</td>
</tr>
<tr>
<td>RTWStateStorageType</td>
<td>Code generation type qualifier</td>
<td>{''}</td>
</tr>
<tr>
<td>Block (Type)/Parameter</td>
<td>Dialog Box Prompt</td>
<td>Values</td>
</tr>
<tr>
<td>------------------------</td>
<td>-------------------</td>
<td>--------</td>
</tr>
<tr>
<td>VectorParams1D</td>
<td>Interpret vector parameters as 1-D</td>
<td>'off'</td>
</tr>
<tr>
<td>ShowAdditionalParam</td>
<td>Show additional parameters</td>
<td>{'off'}</td>
</tr>
<tr>
<td>OutMin</td>
<td>Minimum</td>
<td>{'[ ]'}</td>
</tr>
<tr>
<td>OutMax</td>
<td>Maximum</td>
<td>{'[ ]'}</td>
</tr>
<tr>
<td>OutDataTypeStr</td>
<td>Data type</td>
<td>{'Inherit: auto'}</td>
</tr>
<tr>
<td>LockScale</td>
<td>Lock output data type setting against changes by the fixed-point tools</td>
<td>{'off'}</td>
</tr>
<tr>
<td>SignalType</td>
<td>Signal type</td>
<td>{'auto'}</td>
</tr>
</tbody>
</table>

Data Store Read (DataStoreRead)

**DataStoreElements**
Corresponds to the parameters on the **Element Selection** tab of the block dialog box. See “Specification using the command line”.

**DataStoreName**
Data store name

**SampleTime**
Sample time

Data Store Write (DataStoreWrite)

**DataStoreElements**
Corresponds to the parameters on the **Element Assignment** tab of the block dialog box. See “Specification using the command line”.

**DataStoreName**
Data store name
<table>
<thead>
<tr>
<th>Block (Type)/Parameter</th>
<th>Dialog Box Prompt</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>SampleTime</td>
<td>Sample time (-1 for inherited)</td>
<td>{'-1'}</td>
</tr>
<tr>
<td>Outputs</td>
<td>Number of outputs</td>
<td>{'2'}</td>
</tr>
<tr>
<td>DisplayOption</td>
<td>Display option</td>
<td>'none'</td>
</tr>
<tr>
<td>BusSelectionMode</td>
<td>Bus selection mode</td>
<td>{'off'}</td>
</tr>
<tr>
<td>Environment Controller</td>
<td></td>
<td></td>
</tr>
<tr>
<td>From</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GotoTag</td>
<td>Goto tag</td>
<td>{'A'}</td>
</tr>
<tr>
<td>IconDisplay</td>
<td>Icon display</td>
<td>'Signal name'</td>
</tr>
<tr>
<td>Goto</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GotoTag</td>
<td>Goto tag</td>
<td>{'A'}</td>
</tr>
<tr>
<td>IconDisplay</td>
<td>Icon display</td>
<td>'Signal name'</td>
</tr>
<tr>
<td>TagVisibility</td>
<td>Tag visibility</td>
<td>{'local'}</td>
</tr>
<tr>
<td>Goto Tag Visibility</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GotoTag</td>
<td>Goto tag</td>
<td>{'A'}</td>
</tr>
<tr>
<td>Index Vector</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DataPortOrder</td>
<td>Data port order</td>
<td>{'Zero-based contiguous'}</td>
</tr>
<tr>
<td>Inputs</td>
<td>Number of data ports</td>
<td>{'1'}</td>
</tr>
<tr>
<td>zeroidx</td>
<td>Deprecated in R2010a</td>
<td></td>
</tr>
<tr>
<td>SampleTime</td>
<td>Sample time (-1 for inherited)</td>
<td>{'-1'}</td>
</tr>
<tr>
<td>InputSameDT</td>
<td>Require all data port inputs to have the same data type</td>
<td>{'off'}</td>
</tr>
<tr>
<td>OutMin</td>
<td>Output minimum</td>
<td>{'[]'}</td>
</tr>
<tr>
<td>OutMax</td>
<td>Output maximum</td>
<td>{'[]'}</td>
</tr>
<tr>
<td>Block (Type)/Parameter</td>
<td>Dialog Box Prompt</td>
<td>Values</td>
</tr>
<tr>
<td>------------------------------</td>
<td>--------------------------------------------------------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>OutDataTypeStr</td>
<td>Output data type</td>
<td>{'Inherit: Inherit via internal rule'}</td>
</tr>
<tr>
<td>LockScale</td>
<td>Lock output data type setting against changes by the fixed-point tools</td>
<td>{'off'}</td>
</tr>
<tr>
<td>RndMeth</td>
<td>Integer rounding mode</td>
<td>'Ceiling'</td>
</tr>
<tr>
<td>SaturateOnIntegerOverflow</td>
<td>Saturate on integer overflow</td>
<td>{'off'}</td>
</tr>
<tr>
<td>AllowDiffInputSizes</td>
<td>Allow different data input sizes (Results in variable-size output signal)</td>
<td>{'off'}</td>
</tr>
<tr>
<td>Manual Switch (Manual Switch)</td>
<td>(masked subsystem)</td>
<td></td>
</tr>
<tr>
<td>varsizex</td>
<td>Allow different input sizes (Results in variable-size output signal)</td>
<td>{'off'}</td>
</tr>
<tr>
<td>SampleTime</td>
<td>Sample time (-1 for inherited)</td>
<td>{'-1'}</td>
</tr>
<tr>
<td>Merge (Merge)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inputs</td>
<td>Number of inputs</td>
<td>{'2'}</td>
</tr>
<tr>
<td>InitialOutput</td>
<td>Initial output</td>
<td>{'[]}</td>
</tr>
<tr>
<td>AllowUnequalInputPortWidths</td>
<td>Allow unequal port widths</td>
<td>{'off'}</td>
</tr>
<tr>
<td>InputPortOffsets</td>
<td>Input port offsets</td>
<td>{'[]}</td>
</tr>
<tr>
<td>Multiport Switch (MultiPortSwitch)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Block (Type)/Parameter</td>
<td>Dialog Box Prompt</td>
<td>Values</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>----------------------------------------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>DataPortOrder</td>
<td>Data port order</td>
<td>'Zero-based contiguous'</td>
</tr>
<tr>
<td>Inputs</td>
<td>Number of data ports</td>
<td>{'3'}</td>
</tr>
<tr>
<td>zeroidx</td>
<td>Deprecated in R2010a</td>
<td></td>
</tr>
<tr>
<td>DataPortIndices</td>
<td>Data port indices</td>
<td>{'{1,2,3}' }</td>
</tr>
<tr>
<td>DataPortForDefault</td>
<td>Data port for default case</td>
<td>{'Last data port'}</td>
</tr>
<tr>
<td>DiagnosticForDefault</td>
<td>Diagnostic for default case</td>
<td>'None'</td>
</tr>
<tr>
<td>SampleTime</td>
<td>Sample time (-1 for inherited)</td>
<td>{'-1'}</td>
</tr>
<tr>
<td>InputSameDT</td>
<td>Require all data port inputs to have the same data type</td>
<td>{'off'}</td>
</tr>
<tr>
<td>OutMin</td>
<td>Output minimum</td>
<td>{'[]'}</td>
</tr>
<tr>
<td>OutMax</td>
<td>Output maximum</td>
<td>{'[]'}</td>
</tr>
<tr>
<td>OutDataTypeStr</td>
<td>Output data type</td>
<td>{'Inherit: Inherit via internal rule'}</td>
</tr>
<tr>
<td>LockScale</td>
<td>Lock output data type setting against changes by the fixed-point tools</td>
<td>{'off'}</td>
</tr>
<tr>
<td>RndMeth</td>
<td>Integer rounding mode</td>
<td>'Ceiling'</td>
</tr>
</tbody>
</table>

6-254
### Block-Specific Parameters

<table>
<thead>
<tr>
<th>Block (Type)/Parameter</th>
<th>Dialog Box Prompt</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>SaturateOnIntegerOverflow</td>
<td>Saturate on integer overflow</td>
<td>{'off'}</td>
</tr>
<tr>
<td>AllowDiffInputSizes</td>
<td>Allow different data input sizes (Results in variable-size output signal)</td>
<td>{'off'}</td>
</tr>
</tbody>
</table>

**Mux (Mux)**

| Inputs                         | Number of inputs                                                                 | {'2'}                  |
| DisplayOption                  | Display option                                                                   | 'none' | 'signals' | {'bar'}     |
| UseBusObject                   | For internal use                                                                 |                        |
| BusObject                      | For internal use                                                                 |                        |
| NonVirtualBus                  | For internal use                                                                 |                        |

**Selector (Selector)**

<p>| NumberOfDimensions             | Number of input dimensions                                                       | {'1'}                  |
| IndexMode                      | Index mode                                                                       | 'Zero-based' | {'One-based'} |
| IndexOptionArray               | Index Option                                                                     | 'Select all' | {'Index vector (dialog)'} | 'Index vector (port)'} | 'Starting index (dialog)'} | 'Starting index (port)'} |
| IndexParamArray                | Index                                                                            | cell array            |
| OutputSizeArray                | Output Size                                                                      | cell array            |
| InputPortWidth                 | Input port size                                                                  | {'1'}                 |
| SampleTime                     | Sample time (-1 for inherited)                                                   | {'-1'}                |
| IndexOptions                   | See the IndexOptionArray parameter for more information.                         |                        |
| Indices                        | See the IndexParamArray parameter for more information.                          |                        |</p>
<table>
<thead>
<tr>
<th>Block (Type)/Parameter</th>
<th>Dialog Box Prompt</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>OutputSizes</td>
<td>See the <strong>IndexParamArray</strong> parameter for more information.</td>
<td></td>
</tr>
<tr>
<td>Switch (Switch)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Criteria</td>
<td>Criteria for passing first input</td>
<td>{'u2 &gt;= Threshold'}</td>
</tr>
<tr>
<td>Threshold</td>
<td>Threshold</td>
<td>{'0'}</td>
</tr>
<tr>
<td>ZeroCross</td>
<td>Enable zero-crossing detection</td>
<td>'off'</td>
</tr>
<tr>
<td>SampleTime</td>
<td>Sample time (-1 for inherited)</td>
<td>{'-1'}</td>
</tr>
<tr>
<td>InputSameDT</td>
<td>Require all data port inputs to have the same data type</td>
<td>{'off'}</td>
</tr>
<tr>
<td>OutMin</td>
<td>Output minimum</td>
<td>{'[]}</td>
</tr>
<tr>
<td>OutMax</td>
<td>Output maximum</td>
<td>{'[]}</td>
</tr>
<tr>
<td>OutDataTypeStr</td>
<td>Output data type</td>
<td>{'Inherit: Inherit via internal rule'}</td>
</tr>
<tr>
<td>LockScale</td>
<td>Lock output data type setting against changes by the fixed-point tools</td>
<td>{'off'}</td>
</tr>
<tr>
<td>RndMeth</td>
<td>Integer rounding mode</td>
<td>'Ceiling'</td>
</tr>
<tr>
<td>Block (Type)/Parameter</td>
<td>Dialog Box Prompt</td>
<td>Values</td>
</tr>
<tr>
<td>------------------------</td>
<td>-------------------</td>
<td>--------</td>
</tr>
<tr>
<td>SaturateOnInteger Overflow</td>
<td>Saturate on integer overflow</td>
<td>{'off'}</td>
</tr>
<tr>
<td>AllowDiffInputSizes</td>
<td>Allow different input sizes (Results in variable-size output signal)</td>
<td>{'off'}</td>
</tr>
</tbody>
</table>

**Variant Source (VariantSource)**

| VariantControls | Variant control | {'Variant'} | '(default)' |
| LabelModeActiveChoice | Label mode active choice | {} |
| AllowZeroVariantControls | Allow zero active variant controls | {'off'} | 'on' |
| ShowConditionOnBlock | Show variant condition on block | {'off'} | 'on' |
| GeneratePreprocessorConditionals | Analyze all choices during update diagram and generate preprocessor conditionals | {'off'} | 'on' |
| CompiledActiveVariantControl | | string – '{}' |
|  | | The value is a empty string if no variant is active; or the name of the active variant. Compile the model before querying this property. |
| CompiledActiveVariantPort | | string – {'-1'} |
|  | | The value is -1 if no variant is active; or the index of the active variant. Compile the model before querying this property. |

**Variant Sink (VariantSink)**

<p>| VariantControls | Variant control | {'Variant'} | '(default)' |
| LabelModeActiveChoice | Label mode active choice | {} |
| AllowZeroVariantControls | Allow zero active variant controls | {'off'} | 'on' |</p>
<table>
<thead>
<tr>
<th>Block (Type)/Parameter</th>
<th>Dialog Box Prompt</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>ShowConditionOnBlock</td>
<td>Show variant condition on block</td>
<td>{'off'}</td>
</tr>
<tr>
<td>GeneratePreprocessorConditionals</td>
<td>Analyze all choices during update diagram and generate preprocessor conditionals</td>
<td>{'off'}</td>
</tr>
<tr>
<td>CompiledActiveVariantControl</td>
<td></td>
<td>string — {''}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The value is a empty string if no variant is active; or the name of the active variant. Compile the model before querying this property.</td>
</tr>
<tr>
<td>CompiledActiveVariantPort</td>
<td></td>
<td>string — {'-1'}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The value is -1 if no variant is active; or the index of the active variant. Compile the model before querying this property.</td>
</tr>
</tbody>
</table>

**Vector Concatenate (Concatenate)**

<table>
<thead>
<tr>
<th>NumInputs</th>
<th>Number of inputs</th>
<th>{'2'}</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mode</td>
<td>Mode</td>
<td>{'Vector'}</td>
</tr>
</tbody>
</table>
## Sinks Library Block Parameters

<table>
<thead>
<tr>
<th>Block (Type)/Parameter</th>
<th>Dialog Box Prompt</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Display (Display)</td>
<td>Format</td>
<td></td>
</tr>
<tr>
<td>Numeric display format</td>
<td>Format</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Format</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Format</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Format</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Format</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Format</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Format</td>
<td></td>
</tr>
<tr>
<td>Decimation</td>
<td>Decimation</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Decimation</td>
<td></td>
</tr>
<tr>
<td>Floating Scope (Scope)</td>
<td>Floating display</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Floating display</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Floating display</td>
<td></td>
</tr>
<tr>
<td>SampleTime</td>
<td>Sample time ( -1 for inherited)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Sample time ( -1 for inherited)</td>
<td></td>
</tr>
<tr>
<td>Floating</td>
<td>'off'</td>
<td>{'on'}</td>
</tr>
<tr>
<td>Location</td>
<td>vector — {'[376 294 700 533]'}</td>
<td></td>
</tr>
<tr>
<td>Open</td>
<td>'off'</td>
<td>{'on'}</td>
</tr>
<tr>
<td>NumInputPorts</td>
<td>Do not change this parameter with the command-line. To add inputs, use the signal selector button <img src="image" alt="signal selector button" />.</td>
<td></td>
</tr>
<tr>
<td>TickLabels</td>
<td>'on'</td>
<td>'off'</td>
</tr>
<tr>
<td>ZoomMode</td>
<td>'on'</td>
<td>'xonly'</td>
</tr>
<tr>
<td>AxesTitles</td>
<td>character vector</td>
<td></td>
</tr>
<tr>
<td>Grid</td>
<td>'off'</td>
<td>{'on'}</td>
</tr>
<tr>
<td>TimeRange</td>
<td>{'auto'}</td>
<td></td>
</tr>
<tr>
<td>YMin</td>
<td>{'-5'}</td>
<td></td>
</tr>
<tr>
<td>Block (Type)/Parameter</td>
<td>Dialog Box Prompt</td>
<td>Values</td>
</tr>
<tr>
<td>------------------------</td>
<td>-------------------</td>
<td>----------------------------</td>
</tr>
<tr>
<td>YMax</td>
<td></td>
<td>{'5'}</td>
</tr>
<tr>
<td>SaveToWorkspace</td>
<td></td>
<td>{'off'}</td>
</tr>
<tr>
<td>SaveName</td>
<td></td>
<td>{'ScopeData'}</td>
</tr>
<tr>
<td>DataFormat</td>
<td></td>
<td>{'StructureWithTime'}</td>
</tr>
<tr>
<td>LimitDataPoints</td>
<td></td>
<td>'off'</td>
</tr>
<tr>
<td>MaxDataPoints</td>
<td></td>
<td>{'5000'}</td>
</tr>
<tr>
<td>Decimation</td>
<td></td>
<td>{'1'}</td>
</tr>
<tr>
<td>SampleInput</td>
<td></td>
<td>{'off'}</td>
</tr>
<tr>
<td>SampleTime</td>
<td></td>
<td>{'0'}</td>
</tr>
<tr>
<td>Out1 (Outport)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Port</td>
<td>Port number</td>
<td>{'1'}</td>
</tr>
<tr>
<td>IconDisplay</td>
<td>Icon display</td>
<td>'Signal name'</td>
</tr>
<tr>
<td>BusOutputAsStruct</td>
<td>Output as nonvirtual bus in parent model</td>
<td>{'off'}</td>
</tr>
<tr>
<td>PortDimensions</td>
<td>Port dimensions (-1 for inherited)</td>
<td>{'-1'}</td>
</tr>
<tr>
<td>VarSizeSig</td>
<td>Variable-size signal</td>
<td>{'Inherit'}</td>
</tr>
<tr>
<td>SampleTime</td>
<td>Sample time (-1 for inherited)</td>
<td>{'-1'}</td>
</tr>
<tr>
<td>OutMin</td>
<td>Minimum</td>
<td>{'[]'}</td>
</tr>
<tr>
<td>OutMax</td>
<td>Maximum</td>
<td>{'[]'}</td>
</tr>
<tr>
<td>Block (Type)/Parameter</td>
<td>Dialog Box Prompt</td>
<td>Values</td>
</tr>
<tr>
<td>---------------------------</td>
<td>------------------------------------------------------------------------------------</td>
<td>-----------------------------------------------------------------------</td>
</tr>
<tr>
<td>OutDataTypeStr</td>
<td>Data type</td>
<td>{'Inherit: auto'}</td>
</tr>
<tr>
<td>LockScale</td>
<td>Lock output data type setting against changes by the fixed-point tools</td>
<td>{'off'}</td>
</tr>
<tr>
<td>SignalObject</td>
<td>This parameter does not appear in the block dialog box. Use the Model Data Editor instead. See “Design Data Interface by Configuring Inport and Outport Blocks” (Simulink Coder).</td>
<td>Simulink.Signal object Object of a class that is derived from Simulink.Signal</td>
</tr>
<tr>
<td>StorageClass</td>
<td>This parameter does not appear in the block dialog box. Use the Model Data Editor instead. See “Design Data Interface by Configuring Inport and Outport Blocks” (Simulink Coder).</td>
<td>{'Auto'}</td>
</tr>
<tr>
<td>SignalName</td>
<td>Signal name</td>
<td>character vector</td>
</tr>
<tr>
<td>SignalType</td>
<td>Signal type</td>
<td>{'auto'}</td>
</tr>
<tr>
<td>OutputWhenDisabled</td>
<td>Output when disabled</td>
<td>{'held'}</td>
</tr>
<tr>
<td>InitialOutput</td>
<td>Initial output</td>
<td>{'[]'}</td>
</tr>
<tr>
<td>MustResolveToSignalObject</td>
<td>This parameter does not appear in the block dialog box. Use the Model Data Editor instead. See “For Signals”.</td>
<td>{'off'}</td>
</tr>
<tr>
<td>Block (Type)/Parameter</td>
<td>Dialog Box Prompt</td>
<td>Values</td>
</tr>
<tr>
<td>------------------------</td>
<td>-------------------</td>
<td>--------</td>
</tr>
<tr>
<td>Scope (Scope)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Floating</td>
<td></td>
<td>{'off'}</td>
</tr>
<tr>
<td>Location</td>
<td></td>
<td>vector —</td>
</tr>
<tr>
<td>Open</td>
<td></td>
<td>{'off'}</td>
</tr>
<tr>
<td>NumInputPorts</td>
<td></td>
<td>Do not change this parameter with the <code>set_param</code> command. Instead, use the <code>TimeScopeConfiguration</code> object or <strong>Number of Input Ports</strong> parameter in the Scope block.</td>
</tr>
<tr>
<td>TickLabels</td>
<td></td>
<td>'on'</td>
</tr>
<tr>
<td>ZoomMode</td>
<td></td>
<td>{'on'}</td>
</tr>
<tr>
<td>AxesTitles</td>
<td></td>
<td>character vector</td>
</tr>
<tr>
<td>Grid</td>
<td></td>
<td>'off'</td>
</tr>
<tr>
<td>TimeRange</td>
<td></td>
<td>{'auto'}</td>
</tr>
<tr>
<td>YMin</td>
<td></td>
<td>{'-5'}</td>
</tr>
<tr>
<td>YMax</td>
<td></td>
<td>{'5'}</td>
</tr>
<tr>
<td>SaveToWorkspace</td>
<td></td>
<td>{'off'}</td>
</tr>
<tr>
<td>SaveName</td>
<td></td>
<td>{'ScopeData1'}</td>
</tr>
<tr>
<td>DataFormat</td>
<td></td>
<td>{'StructureWithTime'}</td>
</tr>
<tr>
<td>LimitDataPoints</td>
<td></td>
<td>'off'</td>
</tr>
<tr>
<td>MaxDataPoints</td>
<td></td>
<td>{'5000'}</td>
</tr>
<tr>
<td>Decimation</td>
<td></td>
<td>{'1'}</td>
</tr>
<tr>
<td>SampleInput</td>
<td></td>
<td>{'off'}</td>
</tr>
<tr>
<td>SampleTime</td>
<td></td>
<td>{'0'}</td>
</tr>
<tr>
<td>Block (Type)/Parameter</td>
<td>Dialog Box Prompt</td>
<td>Values</td>
</tr>
<tr>
<td>----------------------------------------</td>
<td>----------------------</td>
<td>---------------------------------------------</td>
</tr>
<tr>
<td>Stop Simulation</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Terminator</td>
<td></td>
<td></td>
</tr>
<tr>
<td>To File (ToFile)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>FileName</strong></td>
<td>File name</td>
<td>{'untitled.mat'}</td>
</tr>
<tr>
<td><strong>MatrixName</strong></td>
<td>Variable name</td>
<td>{'ans'}</td>
</tr>
<tr>
<td><strong>SaveFormat</strong></td>
<td>Save format</td>
<td>{'Timeseries'}</td>
</tr>
<tr>
<td><strong>Decimation</strong></td>
<td>Decimation</td>
<td>{'1'}</td>
</tr>
<tr>
<td><strong>SampleTime</strong></td>
<td>Sample time (-1 for inherited)</td>
<td>{'-1'}</td>
</tr>
<tr>
<td>To Workspace (ToWorkspace)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>VariableName</strong></td>
<td>Variable name</td>
<td>{'simout'}</td>
</tr>
<tr>
<td><strong>MaxDataPoints</strong></td>
<td>Limit data points to last</td>
<td>{'inf'}</td>
</tr>
<tr>
<td><strong>Decimation</strong></td>
<td>Decimation</td>
<td>{'1'}</td>
</tr>
<tr>
<td><strong>SampleTime</strong></td>
<td>Sample time (-1 for inherited)</td>
<td>{'-1'}</td>
</tr>
<tr>
<td><strong>SaveFormat</strong></td>
<td>Save format</td>
<td>{'Timeseries'}</td>
</tr>
<tr>
<td><strong>FixptAsFi</strong></td>
<td>Log fixed-point data as an fi object</td>
<td>{'off'}</td>
</tr>
<tr>
<td>XY Graph (XY scope) (masked subsystem)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>xmin</strong></td>
<td>x-min</td>
<td>{'-1'}</td>
</tr>
<tr>
<td><strong>xmax</strong></td>
<td>x-max</td>
<td>{'1'}</td>
</tr>
<tr>
<td><strong>ymin</strong></td>
<td>y-min</td>
<td>{'-1'}</td>
</tr>
<tr>
<td><strong>ymax</strong></td>
<td>y-max</td>
<td>{'1'}</td>
</tr>
<tr>
<td><strong>st</strong></td>
<td>Sample time</td>
<td>{'-1'}</td>
</tr>
</tbody>
</table>
## Sources Library Block Parameters

<table>
<thead>
<tr>
<th>Block (Type)/Parameter</th>
<th>Dialog Box Prompt</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Band-Limited White Noise</strong> (masked subsystem)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cov</td>
<td>Noise power</td>
<td>{'[0.1]'}</td>
</tr>
<tr>
<td>Ts</td>
<td>Sample time</td>
<td>{'0.1'}</td>
</tr>
<tr>
<td>seed</td>
<td>Seed</td>
<td>{'[23341]'}</td>
</tr>
<tr>
<td>VectorParams1D</td>
<td>Interpret vector parameters as 1-D</td>
<td>'off'</td>
</tr>
<tr>
<td><strong>Chirp Signal</strong> (masked subsystem)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>f1</td>
<td>Initial frequency</td>
<td>{'0.1'}</td>
</tr>
<tr>
<td>T</td>
<td>Target time</td>
<td>{'100'}</td>
</tr>
<tr>
<td>f2</td>
<td>Frequency at target time</td>
<td>{'1'}</td>
</tr>
<tr>
<td>VectorParams1D</td>
<td>Interpret vectors parameters as 1-D</td>
<td>'off'</td>
</tr>
<tr>
<td><strong>Clock</strong> (masked subsystem)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DisplayTime</td>
<td>Display time</td>
<td>{'off'}</td>
</tr>
<tr>
<td>Decimation</td>
<td>Decimation</td>
<td>{'10'}</td>
</tr>
<tr>
<td><strong>Constant</strong> (masked subsystem)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Value</td>
<td>Constant value</td>
<td>{'1'}</td>
</tr>
<tr>
<td>VectorParams1D</td>
<td>Interpret vector parameters as 1-D</td>
<td>'off'</td>
</tr>
<tr>
<td>SampleTime</td>
<td>Sampling time</td>
<td>{'Sample based'}</td>
</tr>
<tr>
<td>OutMin</td>
<td>Output minimum</td>
<td>{'[]'}</td>
</tr>
<tr>
<td>OutMax</td>
<td>Output maximum</td>
<td>{'[]'}</td>
</tr>
<tr>
<td>Block (Type)/Parameter</td>
<td>Dialog Box Prompt</td>
<td>Values</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>-----------------------------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>OutDataTypeStr</td>
<td>Output data type</td>
<td>{'Inherit: Inherit from 'Constant value''}</td>
</tr>
<tr>
<td>LockScale</td>
<td>Lock output data type setting against changes by the fixed-point tools</td>
<td>{'off'}</td>
</tr>
<tr>
<td>SampleTime</td>
<td>Sample time</td>
<td>{'inf'}</td>
</tr>
<tr>
<td>FramePeriod</td>
<td>Frame period</td>
<td>{'inf'}</td>
</tr>
<tr>
<td>Counter Free-Running (Counter Free-Running) (masked subsystem)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NumBits</td>
<td>Number of Bits</td>
<td>{'16'}</td>
</tr>
<tr>
<td>tsamp</td>
<td>Sample time</td>
<td>{'-1'}</td>
</tr>
<tr>
<td>Counter Limited (Counter Limited) (masked subsystem)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>uplimit</td>
<td>Upper limit</td>
<td>{'7'}</td>
</tr>
<tr>
<td>tsamp</td>
<td>Sample time</td>
<td>{'-1'}</td>
</tr>
<tr>
<td>Digital Clock (DigitalClock)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SampleTime</td>
<td>Sample time</td>
<td>{'1'}</td>
</tr>
<tr>
<td>Enumerated Constant (EnumeratedConstant)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OutDataTypeStr</td>
<td>Output data type</td>
<td>{'Enum: SlDemoSign'}</td>
</tr>
</tbody>
</table>
### Block (Type)/Parameter

<table>
<thead>
<tr>
<th>Block (Type)/Parameter</th>
<th>Dialog Box Prompt</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SampleTime</strong></td>
<td>Sample time</td>
<td>{'inf'}</td>
</tr>
<tr>
<td><strong>From File (FromFile)</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>FileName</strong></td>
<td>File name</td>
<td>{'untitled.mat'}</td>
</tr>
<tr>
<td><strong>ExtrapolationBeforeFirstDataPoint</strong></td>
<td>Data extrapolation before first data point</td>
<td>{'Linear extrapolation'}</td>
</tr>
<tr>
<td><strong>InterpolationWithinTime Range</strong></td>
<td>Data interpolation within time range</td>
<td>{'Linear interpolation'}</td>
</tr>
<tr>
<td><strong>ExtrapolationAfterLastDataPoint</strong></td>
<td>Data extrapolation after last data point</td>
<td>{'Linear extrapolation'}</td>
</tr>
<tr>
<td><strong>SampleTime</strong></td>
<td>Sample time</td>
<td>{'0'}</td>
</tr>
<tr>
<td><strong>From Workspace (FromWorkspace)</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>VariableName</strong></td>
<td>Data</td>
<td>{'simin'}</td>
</tr>
<tr>
<td><strong>OutDataTypeStr</strong></td>
<td>Output Data type</td>
<td>{'Inherit: auto'}</td>
</tr>
<tr>
<td><strong>SampleTime</strong></td>
<td>Sample time</td>
<td>{'0'}</td>
</tr>
<tr>
<td><strong>Interpolate</strong></td>
<td>Interpolate data</td>
<td>'off'</td>
</tr>
<tr>
<td><strong>ZeroCross</strong></td>
<td>Enable zero-crossing detection</td>
<td>'off'</td>
</tr>
<tr>
<td><strong>OutputAfterFinalValue</strong></td>
<td>Form output after final data value by</td>
<td>{'Extrapolation'}</td>
</tr>
<tr>
<td>Block (Type)/Parameter</td>
<td>Dialog Box Prompt</td>
<td>Values</td>
</tr>
<tr>
<td>------------------------</td>
<td>-------------------</td>
<td>--------</td>
</tr>
<tr>
<td>In1 (Inport)</td>
<td>Port</td>
<td>{'1'}</td>
</tr>
<tr>
<td></td>
<td>Port number</td>
<td></td>
</tr>
<tr>
<td>IconDisplay</td>
<td>Icon display</td>
<td>'Signal name'</td>
</tr>
<tr>
<td>BusOutputAsStruct</td>
<td>Output as nonvirtual bus</td>
<td>{'off'}</td>
</tr>
<tr>
<td>PortDimensions</td>
<td>Port dimensions (-1 for inherited)</td>
<td>{'-1'}</td>
</tr>
<tr>
<td>VarSizeSig</td>
<td>Variable-size signal</td>
<td>{'Inherit'}</td>
</tr>
<tr>
<td>SampleTime</td>
<td>Sample time (-1 for inherited)</td>
<td>{'-1'}</td>
</tr>
<tr>
<td>OutMin</td>
<td>Minimum</td>
<td>{'[]'}</td>
</tr>
<tr>
<td>OutMax</td>
<td>Maximum</td>
<td>{'[]'}</td>
</tr>
<tr>
<td>OutDataTypeStr</td>
<td>Data type</td>
<td>{'Inherit: auto'}</td>
</tr>
<tr>
<td>LockScale</td>
<td>Lock output data type setting against changes by the fixed-point tools</td>
<td>{'off'}</td>
</tr>
<tr>
<td>Unit</td>
<td>Specify physical unit of the input signal to the block. For a list of acceptable units, see Allowed Units.</td>
<td>{'inherit'}</td>
</tr>
<tr>
<td>Block (Type)/Parameter</td>
<td>Dialog Box Prompt</td>
<td>Values</td>
</tr>
<tr>
<td>------------------------</td>
<td>-------------------</td>
<td>--------</td>
</tr>
<tr>
<td>UnitNoProp</td>
<td>Specify physical unit of the input signal to the block without propagation. For a list of acceptable units, see Allowed Units.</td>
<td>'&lt;Enter unit&gt;'</td>
</tr>
<tr>
<td>SignalType</td>
<td>Signal type</td>
<td>{'auto'}</td>
</tr>
<tr>
<td>LatchByDelayingOutsideSignal</td>
<td>Latch input by delaying outside signal</td>
<td>{'off'}</td>
</tr>
<tr>
<td>LatchInputForFeedbackSignals</td>
<td>Latch input for feedback signals of function-call subsystem outputs</td>
<td>{'off'}</td>
</tr>
<tr>
<td>OutputFunctionCall</td>
<td>Output a function-call trigger signal</td>
<td>{'off'}</td>
</tr>
<tr>
<td>Interpolate</td>
<td>Interpolate data</td>
<td>'off'</td>
</tr>
<tr>
<td>Pulse Generator (DiscretePulseGenerator)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PulseType</td>
<td>Pulse type</td>
<td>{'Time based'}</td>
</tr>
<tr>
<td>TimeSource</td>
<td>Time (t)</td>
<td>{'Use simulation time'}</td>
</tr>
<tr>
<td>Amplitude</td>
<td>Amplitude</td>
<td>{'1'}</td>
</tr>
<tr>
<td>Period</td>
<td>Period</td>
<td>{'10'}</td>
</tr>
<tr>
<td>PulseWidth</td>
<td>Pulse width</td>
<td>{'5'}</td>
</tr>
<tr>
<td>PhaseDelay</td>
<td>Phase delay</td>
<td>{'0'}</td>
</tr>
<tr>
<td>SampleTime</td>
<td>Sample time</td>
<td>{'1'}</td>
</tr>
<tr>
<td>VectorParams1D</td>
<td>Interpret vector parameters as 1-D</td>
<td>'off'</td>
</tr>
<tr>
<td>Ramp (Ramp) (masked subsystem)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>slope</td>
<td>Slope</td>
<td>{'1'}</td>
</tr>
<tr>
<td>start</td>
<td>Start time</td>
<td>{'0'}</td>
</tr>
<tr>
<td>InitialOutput</td>
<td>Initial output</td>
<td>{'0'}</td>
</tr>
<tr>
<td>Block (Type)/Parameter</td>
<td>Dialog Box Prompt</td>
<td>Values</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>--------------------------------------------------------</td>
<td>---------------------------------------------</td>
</tr>
<tr>
<td>VectorParams1D</td>
<td>Interpret vector parameters as 1-D</td>
<td>'off'</td>
</tr>
<tr>
<td>Random Number (RandomNumber)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mean</td>
<td>Mean</td>
<td>{'0'}</td>
</tr>
<tr>
<td>Variance</td>
<td>Variance</td>
<td>{'1'}</td>
</tr>
<tr>
<td>Seed</td>
<td>Seed</td>
<td>{'0'}</td>
</tr>
<tr>
<td>SampleTime</td>
<td>Sample time</td>
<td>{'0.1'}</td>
</tr>
<tr>
<td>VectorParams1D</td>
<td>Interpret vector parameters as 1-D</td>
<td>'off'</td>
</tr>
<tr>
<td>Repeating Sequence (Repeating table) (masked subsystem)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>rep_seq_t</td>
<td>Time values</td>
<td>{'[0 2]'}</td>
</tr>
<tr>
<td>rep_seq_y</td>
<td>Output values</td>
<td>{'[0 2]'}</td>
</tr>
<tr>
<td>Repeating Sequence Interpolated (Repeating Sequence Interpolated) (masked subsystem)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OutValues</td>
<td>Vector of output values</td>
<td>{'[3 1 4 2 1]'}.''</td>
</tr>
<tr>
<td>TimeValues</td>
<td>Vector of time values</td>
<td>{'[0 0.1 0.5 0.6 1]'}.''</td>
</tr>
<tr>
<td>LookUpMeth</td>
<td>Lookup Method</td>
<td>{'Interpolation-Use End Values'}</td>
</tr>
<tr>
<td>tsamp</td>
<td>Sample time</td>
<td>{'0.01'}</td>
</tr>
<tr>
<td>OutMin</td>
<td>Output minimum</td>
<td>{'[]'}</td>
</tr>
<tr>
<td>OutMax</td>
<td>Output maximum</td>
<td>{'[]'}</td>
</tr>
<tr>
<td>OutDataTypeStr</td>
<td>Output data type</td>
<td>'Inherit: Inherit via back propagation'</td>
</tr>
</tbody>
</table>

6-269
<table>
<thead>
<tr>
<th>Block (Type)/Parameter</th>
<th>Dialog Box Prompt</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>OutputDataTypeScaling Mode</td>
<td>Deprecated in R2007b</td>
<td></td>
</tr>
<tr>
<td>OutDataType</td>
<td>Deprecated in R2007b</td>
<td></td>
</tr>
<tr>
<td>OutScaling</td>
<td>Deprecated in R2007b</td>
<td></td>
</tr>
<tr>
<td>LockScale (Deprecated in R2007b)</td>
<td>Lock output data type setting against changes by the fixed-point tools</td>
<td>{'off'}</td>
</tr>
</tbody>
</table>

Repeating Sequence Stair (Repeating Sequence Stair) (masked subsystem)

<table>
<thead>
<tr>
<th>OutValues</th>
<th>Vector of output values</th>
<th>{'[3 1 4 2 1].''}</th>
</tr>
</thead>
<tbody>
<tr>
<td>tsamp</td>
<td>Sample time</td>
<td>{'-1'}</td>
</tr>
<tr>
<td>OutMin</td>
<td>Output minimum</td>
<td>{'[]'}</td>
</tr>
<tr>
<td>OutMax</td>
<td>Output maximum</td>
<td>{'[]'}</td>
</tr>
<tr>
<td>OutDataTypeStr</td>
<td>Output data type</td>
<td>'Inherit: Inherit via back propagation'</td>
</tr>
</tbody>
</table>

<p>| OutputDataTypeScaling Mode             | Deprecated in R2007b                                                              |                             |
| OutDataType                            | Deprecated in R2007b                                                              |                             |
| ConRadixGroup                          | Deprecated in R2007b                                                              |                             |
| OutScaling                             | Deprecated in R2007b                                                              |                             |
| LockScale (Deprecated in R2007b)       | Lock output data type setting against changes by the fixed-point tools           | {'off'} | 'on'          |</p>
<table>
<thead>
<tr>
<th>Block (Type)/Parameter</th>
<th>Dialog Box Prompt</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signal Builder (Sigbuilder block) (masked subsystem)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Signal Editor (SignalEditor)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Filename</td>
<td>File name</td>
<td>{'untitled.mat'}</td>
</tr>
<tr>
<td>ActiveScenario</td>
<td>Active scenario</td>
<td>{'Scenario'}</td>
</tr>
<tr>
<td>ActiveSignal</td>
<td>Signals</td>
<td>{'Signal 1'}</td>
</tr>
<tr>
<td>IsBus</td>
<td>Output a bus signal</td>
<td>'on'</td>
</tr>
<tr>
<td>OutputBusObjectStr</td>
<td>Select bus object</td>
<td>{'Bus: BusObject'}</td>
</tr>
<tr>
<td>SampleTime</td>
<td>Sample time</td>
<td>{'0'}</td>
</tr>
<tr>
<td>Interpolate</td>
<td>Interpolate data</td>
<td>{'off'}</td>
</tr>
<tr>
<td>ZeroCross</td>
<td>Enable zero-crossing detection</td>
<td>{'off'}</td>
</tr>
<tr>
<td>OutputAfterFinalValue</td>
<td>Form output after final data value by</td>
<td>{'Setting to zero'}</td>
</tr>
<tr>
<td>Unit</td>
<td>Unit</td>
<td>{'inherit'}</td>
</tr>
<tr>
<td>PreserveSignalName</td>
<td>—</td>
<td>boolean — {'off'}</td>
</tr>
<tr>
<td>NumberOfScenarios</td>
<td>—</td>
<td>Read-only. Use get_param to get this value.</td>
</tr>
<tr>
<td>NumberOfSignals</td>
<td>—</td>
<td>Read-only. Use get_param to get this value.</td>
</tr>
<tr>
<td>Signal Generator (SignalGenerator)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WaveForm</td>
<td>Wave form</td>
<td>{'sine'}</td>
</tr>
<tr>
<td>TimeSource</td>
<td>Time (t)</td>
<td>{'Use simulation time'}</td>
</tr>
<tr>
<td>Amplitude</td>
<td>Amplitude</td>
<td>{'1'}</td>
</tr>
<tr>
<td>Frequency</td>
<td>Frequency</td>
<td>{'1'}</td>
</tr>
<tr>
<td>Units</td>
<td>Units</td>
<td>'rad/sec'</td>
</tr>
<tr>
<td>VectorParams1D</td>
<td>Interpret vector parameters as 1-D</td>
<td>'off'</td>
</tr>
<tr>
<td>Sine Wave (Sin)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Block (Type)/Parameter</td>
<td>Dialog Box Prompt</td>
<td>Values</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>-------------------</td>
<td>-------------------------</td>
</tr>
<tr>
<td>SineType</td>
<td>Sine type</td>
<td>{'Time based'}</td>
</tr>
<tr>
<td>TimeSource</td>
<td>Time</td>
<td>{'Use simulation time'}</td>
</tr>
<tr>
<td>Amplitude</td>
<td>Amplitude</td>
<td>{'1'}</td>
</tr>
<tr>
<td>Bias</td>
<td>Bias</td>
<td>{'0'}</td>
</tr>
<tr>
<td>Frequency</td>
<td>Frequency</td>
<td>{'1'}</td>
</tr>
<tr>
<td>Phase</td>
<td>Phase</td>
<td>{'0'}</td>
</tr>
<tr>
<td>Samples</td>
<td>Samples per period</td>
<td>{'10'}</td>
</tr>
<tr>
<td>Offset</td>
<td>Number of offset samples</td>
<td>{'0'}</td>
</tr>
<tr>
<td>SampleTime</td>
<td>Sample time</td>
<td>{'0'}</td>
</tr>
<tr>
<td>VectorParams1D</td>
<td>Interpret vector parameters as 1-D</td>
<td>'off'</td>
</tr>
<tr>
<td>Step (Step)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Time</td>
<td>Step time</td>
<td>{'1'}</td>
</tr>
<tr>
<td>Before</td>
<td>Initial value</td>
<td>{'0'}</td>
</tr>
<tr>
<td>After</td>
<td>Final value</td>
<td>{'1'}</td>
</tr>
<tr>
<td>SampleTime</td>
<td>Sample time</td>
<td>{'0'}</td>
</tr>
<tr>
<td>VectorParams1D</td>
<td>Interpret vector parameters as 1-D</td>
<td>'off'</td>
</tr>
<tr>
<td>ZeroCross</td>
<td>Enable zero-crossing detection</td>
<td>'off'</td>
</tr>
<tr>
<td>Uniform Random Number (UniformRandomNumber)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Minimum</td>
<td>Minimum</td>
<td>{'-1'}</td>
</tr>
<tr>
<td>Maximum</td>
<td>Maximum</td>
<td>{'1'}</td>
</tr>
<tr>
<td>Seed</td>
<td>Seed</td>
<td>{'0'}</td>
</tr>
<tr>
<td>SampleTime</td>
<td>Sample time</td>
<td>{'0.1'}</td>
</tr>
<tr>
<td>VectorParams1D</td>
<td>Interpret vector parameters as 1-D</td>
<td>'off'</td>
</tr>
<tr>
<td>Waveform Generator (WaveformGenerator)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Block (Type)/Parameter</td>
<td>Dialog Box Prompt</td>
<td>Values</td>
</tr>
<tr>
<td>------------------------</td>
<td>------------------</td>
<td>--------</td>
</tr>
<tr>
<td>OutMin</td>
<td>Output minimum</td>
<td>{'[]'}</td>
</tr>
<tr>
<td>OutMax</td>
<td>Output maximum</td>
<td>{'[]'}</td>
</tr>
<tr>
<td>OutDataTypeStr</td>
<td>Output data type</td>
<td>'Inherit: Inherit via back propagation'</td>
</tr>
<tr>
<td>LockScale</td>
<td>Lock output data type setting against changes by the fixed-point tools</td>
<td>{'off'}</td>
</tr>
<tr>
<td>RndMeth</td>
<td>Integer rounding mode</td>
<td>'Ceiling'</td>
</tr>
<tr>
<td>SaturateOnIntegerOverflow</td>
<td>Saturate on integer overflow</td>
<td>{'off'}</td>
</tr>
<tr>
<td>SelectedSignal</td>
<td>Output signal</td>
<td>{'1'}</td>
</tr>
<tr>
<td>SampleTime</td>
<td>Sample time</td>
<td>{'0'}</td>
</tr>
</tbody>
</table>
String Library Block Parameters

<table>
<thead>
<tr>
<th>Block (Type)/Parameter</th>
<th>Dialog Box Prompt</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compose String (Compose String)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Format</td>
<td>Format</td>
<td>scalar</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{&quot;&quot;%d %f&quot;&quot;}</td>
</tr>
<tr>
<td>OutDataTypeStr</td>
<td>Output data type</td>
<td>{&quot;stringtype(255)&quot;}</td>
</tr>
<tr>
<td>Scan String (Scan String)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Format</td>
<td>Format</td>
<td>scalar</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{&quot;&quot;%d %f&quot;&quot;}</td>
</tr>
<tr>
<td>String Compare (String Compare)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CaseSensitive</td>
<td>Case sensitivity for string compare</td>
<td>'off'</td>
</tr>
<tr>
<td>CompareOption</td>
<td>Amount of characters to compare</td>
<td>{&quot;Entire string&quot;}</td>
</tr>
<tr>
<td>NumberOfCharacters</td>
<td>Number of characters to compare</td>
<td>{'1'}</td>
</tr>
<tr>
<td>String Concatenate (String Concatenate)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inputs</td>
<td>Number of inputs</td>
<td>{&quot;2&quot;}</td>
</tr>
<tr>
<td>OutDataTypeStr</td>
<td>Output data type</td>
<td>{&quot;stringtype(128)&quot;}</td>
</tr>
<tr>
<td>String Constant (String Constant)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>String</td>
<td>String</td>
<td>{&quot;Hello!&quot;}</td>
</tr>
</tbody>
</table>

6-274
<table>
<thead>
<tr>
<th>Block (Type)/Parameter</th>
<th>Dialog Box Prompt</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>OutDataTypeStr</td>
<td>Output data type</td>
<td>{&quot;stringtype(128)&quot;}</td>
</tr>
</tbody>
</table>

String to ASCII (String to ASCII)

| OutputVectorSize             | Output vector size      | {"31"} | scalar |

String to Double (String to Double)

<table>
<thead>
<tr>
<th>Format</th>
<th>Format</th>
<th>scalar</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>{&quot;&quot;%d %f&quot;&quot;}</td>
<td>&quot;%d&quot;</td>
</tr>
</tbody>
</table>

String to Enum (String to Enum)

| OutDataTypeStr               | Output data type        | {"Enum: SlDemoSign"} | <data type expression> |

String to Single (String to Single)

<table>
<thead>
<tr>
<th>Format</th>
<th>Format</th>
<th>scalar</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>{&quot;&quot;%f&quot;&quot;}</td>
<td>&quot;%d&quot;</td>
</tr>
</tbody>
</table>

Substring (Substring)

<table>
<thead>
<tr>
<th>InheritMaximumLength</th>
<th>Inherit maximum length from input</th>
<th>'off'</th>
<th>{'on'}</th>
</tr>
</thead>
<tbody>
<tr>
<td>OutDataTypeStr</td>
<td>Output data type</td>
<td>{&quot;stringtype(31)&quot;}</td>
<td>&quot;stringtype(N)&quot;</td>
</tr>
<tr>
<td>StringFromIdxToEnd</td>
<td>Output string from 'idx' to end</td>
<td>{'off'}</td>
<td>'on'</td>
</tr>
</tbody>
</table>
### User-Defined Functions Library Block Parameters

<table>
<thead>
<tr>
<th>Block (Type)/Parameter</th>
<th>Dialog Box Prompt</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>MATLAB Function (Stateflow) (masked subsystem)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MATLAB System (MATLABSystem)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>System</td>
<td>System object class name</td>
<td>{''}</td>
</tr>
<tr>
<td>Level-2 MATLAB S-Function (M-S-Function)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FunctionName</td>
<td>S-function name</td>
<td>{'mlfile'}</td>
</tr>
<tr>
<td>Parameters</td>
<td>Arguments</td>
<td>{''}</td>
</tr>
<tr>
<td>Interpreted MATLAB Function (MATLABFcn)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MATLABFcn</td>
<td>MATLAB function</td>
<td>{'sin'}</td>
</tr>
<tr>
<td>OutputDimensions</td>
<td>Output dimensions</td>
<td>{'-1'}</td>
</tr>
<tr>
<td>OutputSignalType</td>
<td>Output signal type</td>
<td>{'auto'}</td>
</tr>
<tr>
<td>Output1D</td>
<td>Collapse 2-D results to 1-D</td>
<td>'off'</td>
</tr>
<tr>
<td>SampleTime</td>
<td>Sample time (-1 for inherited)</td>
<td>{'-1'}</td>
</tr>
<tr>
<td>S-Function (S-Function)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FunctionName</td>
<td>S-function name</td>
<td>{'system'}</td>
</tr>
<tr>
<td>Parameters</td>
<td>S-function parameters</td>
<td>{''}</td>
</tr>
<tr>
<td>SFunctionModules</td>
<td>S-function modules</td>
<td>{''}</td>
</tr>
<tr>
<td>S-Function Builder (S-Function Builder) (masked subsystem)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FunctionName</td>
<td>S-function name</td>
<td>{'system'}</td>
</tr>
<tr>
<td>Parameters</td>
<td>S-function parameters</td>
<td>{''}</td>
</tr>
<tr>
<td>SFunctionModules</td>
<td>S-function modules</td>
<td>{''}</td>
</tr>
</tbody>
</table>
### Additional Discrete Library Block Parameters

<table>
<thead>
<tr>
<th>Block (Type)/Parameter</th>
<th>Dialog Box Prompt</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>State Matrix A</td>
<td>{'[2.6020 -2.2793 0.6708; 1 0 0; 0 1 0]'}</td>
</tr>
<tr>
<td>B</td>
<td>Input Matrix B</td>
<td>{'[ 1; 0; 0]'}</td>
</tr>
<tr>
<td>C</td>
<td>Output Matrix C</td>
<td>{'[0.0184 0.0024 0.0055]'}</td>
</tr>
<tr>
<td>D</td>
<td>Direct Feedthrough Matrix D</td>
<td>{'[0.0033]'}</td>
</tr>
<tr>
<td>InitialCondition</td>
<td>Initial condition for state</td>
<td>{'0.0'}</td>
</tr>
<tr>
<td>InternalDataType</td>
<td>Data type for internal calculations</td>
<td>{'fixdt('double')'}</td>
</tr>
<tr>
<td>StateEqScaling</td>
<td>Scaling for State Equation AX +BU</td>
<td>{'2^0'}</td>
</tr>
<tr>
<td>OutputEqScaling</td>
<td>Scaling for Output Equation CX +DU</td>
<td>{'2^0'}</td>
</tr>
<tr>
<td>LockScale</td>
<td>Lock output data type setting against changes by the fixed-point tools</td>
<td>{'off'}</td>
</tr>
<tr>
<td>RndMeth</td>
<td>Integer rounding mode</td>
<td>'Ceiling'</td>
</tr>
<tr>
<td>DoSatur</td>
<td>Saturate to max or min when overflows occur</td>
<td>{'off'}</td>
</tr>
</tbody>
</table>

### Transfer Fcn Direct Form II Parameters

<table>
<thead>
<tr>
<th>Block (Type)/Parameter</th>
<th>Dialog Box Prompt</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>NumCoefVec</td>
<td>Numerator coefficients</td>
<td>{'[0.2 0.3 0.2]'}</td>
</tr>
<tr>
<td>DenCoefVec</td>
<td>Denominator coefficients excluding lead (which must be 1.0)</td>
<td>{'[-0.9 0.6]'}</td>
</tr>
<tr>
<td>vinit</td>
<td>Initial condition</td>
<td>{'0.0'}</td>
</tr>
<tr>
<td>Block (Type)/Parameter</td>
<td>Dialog Box Prompt</td>
<td>Values</td>
</tr>
<tr>
<td>------------------------</td>
<td>-------------------</td>
<td>--------</td>
</tr>
<tr>
<td>RndMeth</td>
<td>Integer rounding mode</td>
<td>'Ceiling'</td>
</tr>
<tr>
<td>DoSatur</td>
<td>Saturate to max or min when overflows occur</td>
<td>{'off'}</td>
</tr>
</tbody>
</table>

Transfer Fcn Direct Form II Time Varying (masked subsystem)

| vinit                  | Initial condition | {'0.0'} |
| RndMeth                | Integer rounding mode | 'Ceiling' | 'Convergent' | {'Floor'} | 'Nearest' | 'Round' | 'Simplest' | 'Zero' |
| DoSatur                | Saturate to max or min when overflows occur | {'off'} | 'on' |

Unit Delay Enabled (masked subsystem)

| vinit                  | Initial condition | {'0.0'} |
| tsamp                  | Sample time       | {'-1'} |

Unit Delay Enabled External IC (masked subsystem)

| tsamp                  | Sample time       | {'-1'} |

Unit Delay Enabled Resettable (masked subsystem)

| vinit                  | Initial condition | {'0.0'} |
| tsamp                  | Sample time       | {'-1'} |

Unit Delay Enabled Resettable External IC (masked subsystem)

| tsamp                  | Sample time       | {'-1'} |

Unit Delay External IC (masked subsystem)

| tsamp                  | Sample time       | {'-1'} |

Unit Delay Resettable (masked subsystem)

<p>| vinit                  | Initial condition | {'0.0'} |</p>
<table>
<thead>
<tr>
<th>Block (Type)/Parameter</th>
<th>Dialog Box Prompt</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>tsamp</td>
<td>Sample time</td>
<td>{'-1'}</td>
</tr>
</tbody>
</table>

Unit Delay Resettable External IC (Unit Delay Resettable External Initial Condition) (masked subsystem)

| tsamp                  | Sample time       | {'-1'}     |

Unit Delay With Preview Enabled (Unit Delay With Preview Enabled) (masked subsystem)

| vinit                  | Initial condition | {'0.0'}    |
| tsamp                  | Sample time       | {'-1'}     |

Unit Delay With Preview Enabled Resettable (Unit Delay With Preview Enabled Resettable) (masked subsystem)

| vinit                  | Initial condition | {'0.0'}    |
| tsamp                  | Sample time       | {'-1'}     |

Unit Delay With Preview Enabled Resettable External RV (Unit Delay With Preview Enabled Resettable External RV) (masked subsystem)

| vinit                  | Initial condition | {'0.0'}    |
| tsamp                  | Sample time       | {'-1'}     |

Unit Delay With Preview Resettable (Unit Delay With Preview Resettable) (masked subsystem)

| vinit                  | Initial condition | {'0.0'}    |
| tsamp                  | Sample time       | {'-1'}     |

Unit Delay With Preview Resettable External RV (Unit Delay With Preview Resettable External RV) (masked subsystem)

| vinit                  | Initial condition | {'0.0'}    |
| tsamp                  | Sample time       | {'-1'}     |
### Additional Math: Increment - Decrement Library Block Parameters

<table>
<thead>
<tr>
<th>Block (Type)/Parameter</th>
<th>Dialog Box Prompt</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decrement Real World (Real World Value Decrement)</td>
<td>(masked subsystem)</td>
<td></td>
</tr>
<tr>
<td>Decrement Stored Integer (Stored Integer Value Decrement)</td>
<td>(masked subsystem)</td>
<td></td>
</tr>
<tr>
<td>Decrement Time To Zero (Decrement Time To Zero)</td>
<td>(masked subsystem)</td>
<td></td>
</tr>
<tr>
<td>Decrement To Zero (Decrement To Zero)</td>
<td>(masked subsystem)</td>
<td></td>
</tr>
<tr>
<td>Increment Real World (Real World Value Increment)</td>
<td>(masked subsystem)</td>
<td></td>
</tr>
<tr>
<td>Increment Stored Integer (Stored Integer Value Increment)</td>
<td>(masked subsystem)</td>
<td></td>
</tr>
</tbody>
</table>
Mask Parameters

About Mask Parameters

This section lists parameters that describe masked blocks. You can use these descriptive parameters with `get_param` and `set_param` to obtain and specify the properties of a block mask.

The descriptive mask parameters listed in this section apply to all masks, and provide access to all mask properties. Be careful not to confuse these descriptive mask parameters with the mask-specific parameters defined for an individual mask in the Mask Editor Parameters pane.

See “Masking Fundamentals” and “Mask Editor Overview” for information about block masks and the Mask Editor.
### Mask Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description/Prompt</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mask</td>
<td>Turns mask on or off.</td>
<td>{'on'}</td>
</tr>
<tr>
<td>MaskCallbackString</td>
<td>Mask parameter callbacks that are executed when the respective parameter is changed on the dialog. Set by the <strong>Dialog callback</strong> field on the <strong>Parameters</strong> pane of the Mask Editor dialog box. For more information, see “Mask Callback Code”.</td>
<td>pipe-delimited character vector {'' }</td>
</tr>
<tr>
<td>MaskCallbacks</td>
<td>Cell array version of MaskCallbackString.</td>
<td>cell array {'[]'}</td>
</tr>
<tr>
<td>MaskDescription</td>
<td>Block description. Set by the <strong>Mask description</strong> field on the <strong>Documentation</strong> pane of the Mask Editor dialog box.</td>
<td>character vector {''}</td>
</tr>
<tr>
<td>MaskDisplay</td>
<td>Drawing commands for the block icon. Set by the <strong>Icon Drawing commands</strong> field on the <strong>Icon &amp; Ports</strong> pane of the Mask Editor dialog box.</td>
<td>character vector {''}</td>
</tr>
<tr>
<td>MaskEditorHandle</td>
<td>For internal use only.</td>
<td></td>
</tr>
<tr>
<td>MaskEnableString</td>
<td>Option that determines whether a parameter is greyed out in the dialog. Set by the <strong>Enable parameter</strong> check box on the <strong>Parameters</strong> pane of the Mask Editor dialog box.</td>
<td>pipe-delimited character vector {'' }</td>
</tr>
<tr>
<td>MaskEnables</td>
<td>Cell array version of MaskEnableString.</td>
<td>cell array of character vectors, each either 'on' or 'off' {'[]'}</td>
</tr>
<tr>
<td>Parameter</td>
<td>Description/Prompt</td>
<td>Values</td>
</tr>
<tr>
<td>---------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>----------------------</td>
</tr>
<tr>
<td>MaskHelp</td>
<td>Block help. Set by the Mask help field on the Documentation pane of the Mask Editor dialog box.</td>
<td>character vector {''}</td>
</tr>
<tr>
<td>MaskIconFrame</td>
<td>Set the visibility of the icon frame (Visible is on, Invisible is off). Set by the Block Frame option on the Icon &amp; Ports pane of the Mask Editor dialog box.</td>
<td>{'on'}</td>
</tr>
<tr>
<td>MaskIconOpaque</td>
<td>Set the transparency of the icon. Set by the Icon Transparency option on the Icon &amp; Ports pane of the Mask Editor dialog box.</td>
<td>{'opaque'}</td>
</tr>
<tr>
<td>MaskIconRotate</td>
<td>Set the rotation of the icon (Rotates is on, Fixed is off). Set by the Icon Rotation option on the Icon &amp; Ports pane of the Mask Editor dialog box.</td>
<td>'on'</td>
</tr>
<tr>
<td>MaskIconUnits</td>
<td>Set the units for the drawing commands. Set by the Icon Units option on the Icon &amp; Ports pane of the Mask Editor dialog box.</td>
<td>'pixel'</td>
</tr>
<tr>
<td>MaskInitialization</td>
<td>Initialization commands. Set by the Initialization commands field on the Initialization pane of the Mask Editor dialog box.</td>
<td>MATLAB command {''}</td>
</tr>
<tr>
<td>MaskNames</td>
<td>Cell array of mask dialog parameter names. Set inside the Variable column in the Parameters pane of the Mask Editor dialog box.</td>
<td>matrix {'[]}</td>
</tr>
<tr>
<td>Parameter</td>
<td>Description/Prompt</td>
<td>Values</td>
</tr>
<tr>
<td>--------------------------</td>
<td>------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>---------------------------------------------</td>
</tr>
<tr>
<td>MaskPortRotate</td>
<td>Specify the port rotation policy for the masked block. Set in the <strong>Port Rotation</strong> area on the <strong>Icon &amp; Ports</strong> pane of the Mask Editor dialog box.</td>
<td>{'default'</td>
</tr>
<tr>
<td></td>
<td>For more information, see “Adjust Visual Presentation to Improve Model Readability”.</td>
<td></td>
</tr>
<tr>
<td>MaskPrompts</td>
<td>List of dialog parameter prompts (see below). Set inside the <strong>Dialog parameters</strong> area on the <strong>Parameters</strong> pane of the Mask Editor dialog box.</td>
<td>cell array of character vectors [ [] ]</td>
</tr>
<tr>
<td>MaskPromptString</td>
<td>List of dialog parameter prompts (see below). Set inside the <strong>Dialog parameters</strong> area on the <strong>Parameters</strong> pane of the Mask Editor dialog box.</td>
<td>character vector { '' }</td>
</tr>
<tr>
<td>MaskPropertyName</td>
<td>Pipe-delimited version of MaskNames.</td>
<td>character vector { '' }</td>
</tr>
<tr>
<td>MaskRunInitForIconRedraw</td>
<td>Specifies whether Simulink must run mask initialization before executing the mask icon commands.</td>
<td>{'auto'}</td>
</tr>
<tr>
<td>MaskSelfModifiable</td>
<td>Indicates that the block can modify itself. Set by the <strong>Allow library block to modify its contents</strong> check box on the <strong>Initialization</strong> pane of the Mask Editor dialog box.</td>
<td>'on'</td>
</tr>
<tr>
<td>Parameter</td>
<td>Description/Prompt</td>
<td>Values</td>
</tr>
<tr>
<td>----------------------</td>
<td>-------------------------------------------------------------------------------------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>MaskStyles</td>
<td>Determines whether the dialog parameter is a check box, edit field, or pop-up list. Set by the <strong>Type</strong> column in the <strong>Parameters</strong> pane of the Mask Editor dialog box.</td>
<td>cell array {'[]'} edit, checkbox, popup, combobox, radiobutton, slider, dial, spinbox, unit, textarea, customtable, unidt, min, max.</td>
</tr>
<tr>
<td>MaskStyleString</td>
<td>Comma-separated version of <strong>MaskStyles</strong>.</td>
<td>character vector {''}</td>
</tr>
<tr>
<td>MaskTabNameString</td>
<td>For internal use only.</td>
<td></td>
</tr>
<tr>
<td>MaskTabNames</td>
<td>For internal use only.</td>
<td></td>
</tr>
<tr>
<td>MaskToolTipsDisplay</td>
<td>Determines which mask dialog parameters to display in the tooltip for this masked block. Specify as a cell array of 'on' or 'off' values, each of which indicates whether to display the parameter named at the corresponding position in the cell array returned by <strong>MaskNames</strong>.</td>
<td>cell array of 'on' and 'off' {&quot;}</td>
</tr>
<tr>
<td>MaskToolTipString</td>
<td>Comma-delimited version of <strong>MaskToolTipsDisplay</strong>.</td>
<td>character vector {''}</td>
</tr>
<tr>
<td>MaskTunableValues</td>
<td>Allows the changing of mask dialog values during simulation. Set by the <strong>Tunable</strong> column in the <strong>Parameters</strong> pane of the Mask Editor dialog box.</td>
<td>cell array of character vectors {'[]'}</td>
</tr>
<tr>
<td>MaskTunableValueString</td>
<td>Comma-delimited character vector version of <strong>MaskTunableValues</strong>.</td>
<td>delimited character vector {''}</td>
</tr>
<tr>
<td>MaskType</td>
<td>Mask type. Set by the <strong>Mask type</strong> field on the <strong>Documentation</strong> pane of the Mask Editor dialog box.</td>
<td>character vector {'Stateflow'}</td>
</tr>
<tr>
<td>MaskValues</td>
<td>Dialog parameter values.</td>
<td>cell array {'[]'}</td>
</tr>
<tr>
<td>Parameter</td>
<td>Description/Prompt</td>
<td>Values</td>
</tr>
<tr>
<td>-----------------------</td>
<td>------------------------------------------------------------------------------------</td>
<td>-------------------------------</td>
</tr>
<tr>
<td>MaskValueString</td>
<td>Delimited character vector version of MaskValues.</td>
<td>delimited character vector {''}</td>
</tr>
<tr>
<td>MaskVarAliases</td>
<td>Specify aliases for a block's mask parameters. The aliases must appear in the same order as the parameters appear in the block's MaskValues parameter.</td>
<td>cell array {'[ ]'}</td>
</tr>
<tr>
<td>MaskVarAliasString</td>
<td>For internal use only.</td>
<td></td>
</tr>
<tr>
<td>MaskVariables</td>
<td>List of the dialog parameters' variables (see below). Set inside the Dialog parameters area on the Parameters pane of the Mask Editor dialog box.</td>
<td>character vector {''}</td>
</tr>
<tr>
<td>MaskVisibilityString</td>
<td>Specifies visibility of parameters. Set with the Show parameter check box in the Options for selected parameter area on the Parameters pane of the Mask Editor dialog box.</td>
<td>matrix {'[]'}</td>
</tr>
<tr>
<td>MaskVisibilityString</td>
<td>Delimited character vector version of MaskVisibilityString.</td>
<td>character vector {''}</td>
</tr>
<tr>
<td>MaskWSVariables</td>
<td>List of the variables defined in the mask workspace (read only).</td>
<td>matrix {'[]'}</td>
</tr>
</tbody>
</table>

See Control Masks Programmatically, for more information on setting the mask parameters from the MATLAB command line.
Tools and Apps — Alphabetical List
Bus Editor

Create, modify, and manage Simulink.Bus objects

Description

To interactively create a Simulink.Bus object, and the Simulink.BusElement objects that it contains, use the Simulink Bus Editor.

The Bus objects that you create using the Bus Editor are stored in the MATLAB base workspace. To simulate a block that uses a Bus object, the Bus object must be in the base workspace or in a data dictionary.

To save Bus object definitions using the Bus Editor, export the Bus objects from the base workspace into a MATLAB code file or MAT-file. If you save a Bus object in a MATLAB code file, you can edit it programmatically. For details, see “Create Bus Objects Programmatically”.

You can use the Bus Editor to change and delete existing Bus objects and bus elements. The Bus Editor displays all Bus objects in the base workspace. Changes that create, reorder, or delete entities take effect immediately in the base workspace.

Note The Bus Editor does not provide an undo capability. Changes to properties take effect when you apply them.

For more information on Bus objects, see “Specify Bus Properties with Bus Objects”.

Open the Bus Editor

- On the Modeling tab, in the Design gallery, click Bus Editor.
- At the MATLAB command prompt, enter buseditor.
- In the Model Explorer, in a Bus object dialog box, click the Launch Bus Editor button.
Examples

Create Bus Objects Using the Bus Editor

This example shows how to create a Bus object that corresponds to the control bus in model. This bus contains two signals as elements.

1. In the Bus Editor, click the **Add Bus** button.

   A Bus object with a default name appears in the left pane and its default properties appear in the **Simulink.Bus** dialog pane.

2. Specify the name for the Bus object using the **Name** property. For easy identification, use a name that reflects the bus associated with the bus object. The bus that combines the valve inputs is named control, therefore, name the Bus object CONTROL.

   **Tip** Click **Apply** after editing properties or choose to automatically apply changes when you click a different object.

3. Click the **Add/Insert BusElement** button.

   A bus element with a default name and default properties is created in the CONTROL Bus object. The bus element appears in the left pane nested under the CONTROL Bus object.
4 In the **Simulink.BusElement** dialog pane, set **Name** to VALVE1, which reflects the name of the corresponding signal in the bus.

5 To determine the signal properties for **BusElement** object VALVE1, open the Block Parameters dialog box for the Constant block.

The properties on the **Signal Attributes** tab match the default **BusElement** object properties, so no change is required.

6 Similarly, create a **BusElement** object named VALVE2 that corresponds to the valve2 element.

7 To determine the signal properties for **BusElement** object VALVE2, open the Block Parameters dialog box for the Constant1 block.

The **Output data type** specified by the Constant1 block is int8, which does not match the default **BusElement** object properties. Set the **DataType** property of **BusElement** object VALVE2 to int8.

**Nest Bus Objects Using the Bus Editor**

Any signal in a bus can be another bus, which can in turn contain subordinate buses, to any depth. Describing nested buses using **Bus** objects requires nesting the **Bus** object definitions.

Model has two buses, with one bus nested inside the other bus.

This example shows how to nest a **Bus** object that corresponds to the *control* bus in a **Bus** object that corresponds to the *main* bus.
1. Create a Bus object named CONTROL that fully specifies the control bus.

2. Create a Bus object named MAIN that fully specifies the main bus, temporarily using default values for the BusElement object that specifies the nested control bus.

To identify that this BusElement object corresponds to a nested bus, you can name it VALVES_BUS.

3. In the Simulink.BusElement dialog pane for VALVES_BUS, set the Data type parameter to Bus: <object name>, where <object name> is the name of the Bus object for the control bus.

Tip: A Bus object cannot directly or indirectly reference itself. If you define a circular structure, the Bus Editor warns you and sets the data type of the element that would have completed the circle to double.

Edit Bus Objects Using the Bus Editor

To edit a Bus object or BusElement object:

1. In the left pane, select the object to edit.
2 In the Simulink.Bus or Simulink.BusElement dialog pane, edit the object properties.
3 Click Apply.

When you have a Bus object selected, you can edit its BusElement objects in the middle pane.

To make the same edit to a property in multiple BusElement objects, use the middle pane. Hold the Ctrl key while you select multiple bus element objects and enter the new value in the property for one of the selected bus element objects. The change applies to all the selected BusElement objects.

You can use the Bus Editor to copy and paste Bus objects or BusElement objects. Right-click an object and use the context menu to copy and paste the Bus object. Copying a Bus object also copies its BusElement objects. The copied objects have the same property values as the original objects.

To change the order of bus elements in a Bus object, in the left pane, select a bus element and use the Move Element Up or Move Element Down button.

To delete a Bus object, in the left pane, select the Bus object and click the Delete button or Delete key. Deleting a Bus object deletes its BusElement objects.

If you delete a Bus object, in the Simulink Editor, update any blocks that use that Bus object. To find where a bus object is used in a model, see “Finding Blocks That Use a Specific Variable”.

Save Bus Objects Using the Bus Editor

When you create Bus objects in the base workspace, before you close MATLAB, save the Bus objects for future use.

You can save Bus objects to one of these locations:

- Data dictionary
- MATLAB code file
- MAT-file
• Database or other external files

The Bus Editor can save Bus objects in a MAT-file or MATLAB code file. To save Bus objects in other locations, see “Save Bus Objects”.

**Tip** To display a reminder to save Bus objects before closing the Bus Editor, select Options > Always Warn Before Closing.

1. Select which Bus objects you want to export.
   - To export all visible Bus objects in the Bus Editor, select File > Export to File. Bus objects hidden by a filter do not export.
   - To export only selected Bus objects, right-click a Bus object, then select Export to File.
   - To include nested Bus objects used by the selected Bus object, select Export with Dependent Bus Objects to File.

2. In the Export dialog box, specify the name for the export file.
3. Set Save as type to MATLAB files (*.m) or MAT-files (*.mat).
4. Click Save.
5. For a MATLAB code file, in the additional dialog box that appears, select the export format.
   - Cell — Stores Bus objects in a compressed format.
   - Object — Store Bus objects in the Simulink.Bus object format, which is easier to read and edit.

**Filter Displayed Bus Objects in the Bus Editor**

By default, the Bus Editor displays all Bus objects that exist in the base workspace, in alphabetical order. When you want to reduce the number of Bus objects displayed, you can apply a filter.

The Bus Editor can display only those Bus objects that have:

- Names that match a given search term or regular expression
- A specified relationship to a specified Bus object
You can work with any Bus object that is visible in a filtered display exactly as in an unfiltered display. If you change the name or dependency of an object so that it no longer passes the current filter, the object disappears from the display. Conversely, if some activity outside the Bus Editor changes a filtered object so that it passes the current filter, the object immediately becomes visible.

**Tip** Bus objects created or imported from outside the Bus Editor are not visible until you reopen the Bus Editor.

Operations performed in the left pane for the base workspace affect only visible objects. An object that a filter hides is unaffected by the operation. To act on all existing Bus objects, clear the filter.

In this example, the Bus Editor displays the Bus objects that are in the base workspace.

These Bus objects form two hierarchies. A1_Top is the parent of A2_Mid, which is the parent of A3_Bot. Similarly, B1_Top is the parent of B2_Mid, which is the parent of B3_Bot.

To filter by name, select by Bus Name from the list and type the filter term. See “Regular Expressions” (MATLAB) for complete information about MATLAB regular expression syntax.

As you type, the Bus Editor updates dynamically to show only the Bus objects whose names match the filter expression. The comparison is case-sensitive. For example, typing A displays only the Bus objects whose names contain an uppercase A.
To filter by relationship, select by Bus Object Dependency from the list. In the second list that appears, select the type of relationship to display.

- **Bus Object and Parents** — Show a specified bus object and all superior Bus objects in the hierarchy (default).
- **Bus Object and Dependents** — Show a specified Bus object and all subordinate Bus objects in the hierarchy.
- **Bus Object and Related Objects** — Show a specified Bus object and all superior and subordinate Bus objects.

Type a Bus object by name or select an existing Bus object name from the list.

For example, if you select Bus Object and Dependents and type B2, the Bus Editor displays only the Bus object hierarchies that contain the B2 Bus object.

---

**Load Bus Objects Into the Base Workspace**

You can use the Bus Editor to import the definitions from a MAT-file or MATLAB code file to the base workspace. The import loads the complete contents of the file, not just the Bus objects.
1. Choose **File > Import into Base Workspace**.
2. Use the Open File dialog box to navigate to and import the desired file.

**See Also**

**Blocks**
Bus Creator | Out Bus Element

**Classes**
Simulink.Bus | Simulink.BusElement

**Topics**
“Specify Bus Properties with Bus Objects”
“Create Bus Objects Programmatically”

**Introduced before R2006a**
Simulation Data Inspector

Inspect and compare data and simulation results to validate and iterate model designs

Description
The Simulation Data Inspector visualizes and compares multiple kinds of data.

Using the Simulation Data Inspector, you can inspect and compare time series data at multiple stages of your workflow. This example workflow shows how the Simulation Data Inspector supports all stages of the design cycle:

1 “View Data with the Simulation Data Inspector”.

Run a simulation in a model configured to log data to the Simulation Data Inspector, or import data from the workspace or a MAT-file. You can view and verify model input data or inspect logged simulation data while iteratively modifying your model diagram, parameter values, or model configuration.

2 “Inspect Simulation Data”.

Plot signals on multiple subplots, zoom in and out on specified plot axes, and use data cursors to understand and evaluate the data. “Create Plots Using the Simulation Data Inspector” to tell your story.

3 “Compare Simulation Data”

Compare individual signals or simulation runs and analyze your comparison results with relative, absolute, and time tolerances. The compare tools in the Simulation Data Inspector facilitate iterative design and allow you to highlight signals that do not meet your tolerance requirements. For more information about the comparison operation, see “How the Simulation Data Inspector Compares Data”.

4 “Save and Share Simulation Data Inspector Data and Views”.

Share your findings with others by saving Simulation Data Inspector data and views.

You can also harness the capabilities of the Simulation Data Inspector from the command line. For more information, see “Inspect and Compare Data Programmatically”.
Open the Simulation Data Inspector

- Simulink Toolstrip: On the Simulation tab, under Review Results, click Data Inspector.
- Click the streaming badge on a signal to open the Simulation Data Inspector and plot the signal.
- MATLAB command prompt: Enter `Simulink.sdi.view`.

Examples

Add Signals to a Run

This example shows how to use `Simulink.sdi.createRunOrAddToStreamedRun` to add data to an existing run for a model. In this example, you add logged states data to the run created through simulation.

Simulate the Model

Simulate the model to generate data. The model `slexAircraftExample` is configured to log outputs, so the Simulation Data Inspector automatically creates a run with the logged outport data. Using this simulation syntax, `out` contains the outport data (`yout`) and the states data (`xout`).

```matlab
load_system('slexAircraftExample')
out = sim('slexAircraftExample','ReturnWorkspaceOutputs','on',... 'SaveFormat','Dataset');
```

Add Logged States Data to Run

The Simulation Data Inspector automatically created a run for the logged outport data. Add the logged state data to the existing run using `Simulink.sdi.createRunOrAddToStreamedRun`.

```matlab
Simulink.sdi.createRunOrAddToStreamedRun('slexAircraftExample','Run 1',... {'out'}, {out});
```

Open the Simulation Data Inspector to View Results

Using `Simulink.sdi.createRunOrAddToStreamedRun` avoids redundancy in the data shown in the Simulation Data Inspector. Using `Simulink.sdi.createRun` to bring the
states data into the Simulation Data Inspector creates a second run. 
Simulink.sdi.addToRun creates a duplicate signal from the outport data. Using 
Simulink.sdi.createRunOrAddToStreamedRun, you can include all simulation data 
in a single run without duplicating any signals.

Simulink.sdi.view

Modify Parameter for Several Runs
This example shows how to modify a parameter for all the runs in the Simulation Data 
Inspector programmatically.

Generate Runs
Load the vdp model and mark the x1 and x2 signals for logging. Then, run several 
simulations.

% Clear all data from the Simulation Data Inspector repository
Simulink.sdi.clear

% Load the model and mark signals of interest for streaming
load_system('vdp')
Simulink.sdi.markSignalForStreaming('vdp/x1',1,'on')
Simulink.sdi.markSignalForStreaming('vdp/x2',1,'on')

% Simulate the model with several Mu values
for gain = 1:5
    gainVal = num2str(gain);
    set_param('vdp/Mu','Gain',gainVal)
sim('vdp');
end

Use Simulink.sdi.getRunCount to Assign Tolerance to x1 Signals

count = Simulink.sdi.getRunCount;

for a = 1:count
    runID = Simulink.sdi.getRunIDByIndex(a);
vdpRun = Simulink.sdi.getRun(runID);
sig = vdpRun.getSignalByIndex(1);
sig.AbsTol = 0.1;
end
% Open the Simulation Data Inspector to view your data
Simulink.sdi.view

- “View Data with the Simulation Data Inspector”
- “Inspect Simulation Data”
- “Compare Simulation Data”
- “Iterate Model Design Using the Simulation Data Inspector”

**Programmatic Use**

Simulink.sdi.view opens the Simulation Data Inspector from the MATLAB command line.

**See Also**

**Functions**

Simulink.sdi.clear | Simulink.sdi.clearPreferences | Simulink.sdi.snapshot

**Topics**

“View Data with the Simulation Data Inspector”
“Inspect Simulation Data”
“Compare Simulation Data”
“Iterate Model Design Using the Simulation Data Inspector”

**Introduced in R2010b**
Simulink Onramp

Three-hour interactive training course included with Simulink license

Description
To help you get started quickly with Simulink basics, Simulink Onramp provides a self-paced, interactive tutorial.

After completing Simulink Onramp, you will be able to create, modify, and troubleshoot Simulink models that simulate dynamic systems and perform basic signal analysis.

To teach concepts incrementally, Simulink Onramp uses hands-on exercises. You receive automated assessments and feedback after submitting tasks. Your progress is saved if you exit the application, so you can complete the training in multiple sessions.

Simulink Onramp covers these topics:

- Simulink blocks and signals
- Visualizing signal values during simulation
- Math and logic operators for algorithms
- Obtaining help from documentation
- MATLAB variables and functions in Simulink
- Dynamic systems in Simulink
- Discrete-time systems
- Continuous-time systems
- Simulation duration

Simulink Onramp helps you practice what you learn with these projects:

- Automotive Performance Modes
- Modeling a Thermostat
- Peregrine Falcon Dive
Open the Simulink Onramp

• On the Simulink Start Page, click the Simulink Onramp button.
• On the Simulink Toolstrip quick access toolbar, click Help > Learn Simulink.
• At the MATLAB command prompt, enter learning.simulink.launchOnramp('simulink').

See Also

Topics
“Simulink Block Diagrams”
“Create a Simple Model”
“Navigate Model”
“Model-Based Design with Simulink”
“Build and Edit a Model in the Simulink Editor”
“Simulink Models” on page 13-7

Introduced in R2019a
Solver Profiler

Identify solver performance bottlenecks

**Description**

Use the **Solver Profiler** to examine solver and model behavior to identify issues that can contribute to poor simulation performance.

Use the **Solver Profiler** to analyze a model for patterns that affect its simulation. The **Solver Profiler** presents graphical and statistical information about the simulation, solver settings, events, and errors. You can use this data to identify locations in the model that caused simulation bottlenecks.

In addition, there are multiple factors that can limit the simulation speed. The Solver Profiler logs and reports all the major events that occur when simulating a model:

- Zero-crossing events
- Solver exception events
- Solver reset events
- Jacobian computation events

These events are common and necessary for an accurate simulation. However, they do incur computational cost and frequent occurrences can slow down or even stall the simulation.

**Available actions**

- Trace
- Filter
- Explore

**Information Panes**

- Statistics
- Step Size
• Suggestions

Open the Solver Profiler

Open the Solver Profiler by clicking the hyperlink in the lower-right corner of the Simulink Editor.

Examples

Locate source blocks for solver events

This example shows how to find the source of a particular solver event using a model that simulates a ball bouncing off the ground. As the ball comes to rest, the density of zero-crossings increase, leading to a solver slowdown.

To begin, open the ex_spi_show_zc model and launch the Solver Profiler.

Click Run to profile the model.
Notice that the model has 45 zero-crossing events, Select the **Zero Crossing** tab in the Results pane.

Select the `ex_spi_show_zc/Switch` field in the Results pane. All zero crossing events caused by the Switch block are highlighted in the **Step Size** plot.

In the **Trace** section of the toolstrip, select **Highlight Block**.

You should see the Switch block causing the zero crossing events.
“Examine Model Dynamics Using Solver Profiler”
“Solver Resets”
“Zero-Crossing Events”
“Solver Exception Events”
“Jacobian Logging and Analysis”

**Parameters**

**Log**

*States & Zero Crossing — Save model states to file*

off (default) | on

By default, the profiler does not save the states of the model. Enabling this parameter configures the profiler to save the states to a MAT-file.

*Simscape States — Save Simscape states to file*

off (default) | on

Enable this parameter to save Simscape states to a file.

*Model Jacobian — Save model Jacobian*

off (default) | on

Option to log the solver Jacobian matrices to memory. This option is useful for simulations that use implicit solvers. For a comparison of solvers, see “Compare Solvers”.

---

7-20
For a more detailed explanation of the solver Jacobian, see “Choose a Jacobian Method for an Implicit Solver”.

**From — Profiler start time**
model start time (default) | scalar

Time, in seconds, of the simulation that the profiler starts analyzing the model. This is not the same as the start time of the simulation.

**To — Profiler stop time**
model stop time (default) | scalar

Time, in seconds, of the simulation to which the profiler should profile the model. By default, the analysis continues until the end of the simulation. Changing this parameter does not change the stop time of the model which you specify in the Model Configuration Parameters.

A value less than the configured stop time of the model stops the profiling and simulation at StopTime.

**Buffer — Memory impact of logging**
50000 (default) | positive scalar

Maximum number of events that are logged. If the number of logged events reaches this value and memory is available, increase BufferSize. If memory is limited, consider lowering the value.

**Filter**

**Solver Exception — Show solver exception events**
off (default) | on

Enable this parameter to highlight solver exception events in the Step Size plot of the Solver Profiler.

**Solver Reset — Show solver reset events**
off (default) | on

Enable this parameter to highlight solver reset events in the Step Size plot of the Solver Profiler.

**Zero Crossing — Show zero crossing events**
off (default) | on
Enable this parameter to highlight zero crossing events in the Step Size plot of the Solver Profiler.

**Jacobian Update — Show Jacobian update events**  
off (default) | on

Enable this parameter to highlight Jacobian update events in the Step Size plot of the Solver Profiler.

**Rule Customization — Change thresholds for profiler rules**  
off (default) | on

Click **Rule Customization** in the Solver Profiler to access the rule set. You can change the thresholds for most of these rules and also select which rules you want to apply selectively during a simulation run.

To modify a rule, enable its customization and then enter a desired threshold value.
Custom Rule Set

You can override the settings on the Rule Set dialog box by specifying a custom rule set.

Create a rule set as a MATLAB script and specify the path to the script in the Custom Rule Set section of the Rule Set dialog box.

A simple rule set example looks as follows:

```matlab
function diagnosticsString = customRule(profilerData)
    if isempty(profilerData.zcEvents)
        diagnosticsString{1} = 'No zero crossing event detected.';
    else
        diagnosticsString{1} = 'Zero-crossing events detected.';
    end
end
```

The input to the function is an array of structures called profilerData. This array of structures organizes all the information that the Solver Profiler collects during a profiling run. It contains the following substructures.

<table>
<thead>
<tr>
<th>Substructure</th>
<th>Fields</th>
</tr>
</thead>
</table>
| stateInfo: Stores information on block states | • name: Block name  
• value: State values  
• blockIdx: Block ID |
| blockInfo: Cross-reference of blocks and state IDs | • name: Block name  
• stateIdx: State ID |
| zcSrcInfo: Stores information on blocks causing zero crossing events | • name: Block name  
• blockIdx: Block ID |
| zcEvents: Cross-reference of the time stamps of zero crossing events and the corresponding state IDs | • t: Event timestamp  
• srcIdx: Block ID |
| exceptionEvents: Cross-reference of exception event timestamps, the ID of the corresponding state that caused the event, and the cause. | • t: Event timestamp  
• stateIdx: State ID  
• cause: Cause of exception |
| resetTime: Stores timestamps of solver resets. | None |
### Substructure

<table>
<thead>
<tr>
<th>Fields</th>
<th>Data Types: double</th>
</tr>
</thead>
<tbody>
<tr>
<td>tout: Stores simulation times.</td>
<td></td>
</tr>
</tbody>
</table>

### More About

## Statistics Pane

The statistics pane displays information on model parameters, including:

- **Average step size** — A measure of how fast the solver advances. It is calculated as the total simulation time divided by the number of steps the solver used. It is bounded by the model configuration parameters **Max step size** and **Min step size**.
- **Max step size usage** — The percentage of maximum step sizes used by the solver among all step sizes.
- **Zero crossing** — A solver-specific event that affects model dynamics. During simulation, the solver detects zero crossing. Zero crossing detection incurs computation cost. For more information, see “Zero-Crossing Detection”.
- **Solver reset** — An event that causes the solver to reset its parameters. Solver reset detection incurs computation cost. The solver reset statistics are broken down into **Zero Crossing, Discrete Signal, ZOH Signal, Block Change, Initial Reset, and Internal** solver reset events. For more information, see “Solver Resets”.
- **Solver exception** — An event that renders the solver unable to solve model states to meet accuracy specifications. To solve model states accurately, the solver has to run a few adjusted trials, which incur computation cost.
- **Error control exception** — An event where a solution obtained by the solver has an error that is greater than the tolerance specification.
- **Newton iteration exception** — An event specific to implicit solvers. Newton iterations do not converge after a few trials.
- **Infinite state exception** — An event where one or more states solved by the solver are infinite.
- **DAE newton iteration exception** — An event specific to implicit solvers for Simscape models. The Newton iteration does not converge even though the solver violates the minimum step size constraint.
Suggestions and Exceptions Pane

The suggestions and exceptions pane displays information on exceptions, including:

- **Zero crossing** — A solver-specific event that affects model dynamics. During simulation, the solver detects zero crossing. Zero crossing detection incurs computation cost. For more information, see “Zero-Crossing Detection”.
- **Solver reset** — An event that causes the solver to reset its parameters. Solver reset detection incurs computation cost. The solver reset statistics are broken down into Zero Crossing, Discrete Signal, ZOH Signal, Block Change, Initial Reset, and Internal solver reset events. For more information, see “Solver Resets”.
- **Solver exception** — An event that renders the solver unable to solve model states to meet accuracy specifications. To solve model states accurately, the solver has to run a few adjusted trials, which incur computation cost.
- **Error control exception** — An event where a solution obtained by the solver has an error that is greater than the tolerance specification.
- **Newton iteration exception** — An event specific to implicit solvers. Newton iterations do not converge after a few trials.
- **Infinite state exception** — An event where one or more states solved by the solver are infinite.
- **DAE newton iteration exception** — An event specific to implicit solvers for Simscape models. The Newton iteration does not converge even though the solver violates the minimum step size constraint.

Solver Reset Pane

The **Solver Reset** pane shows a tabulated list of solver resets that have occurred when simulating the model. Each row of the table corresponds to a block in the model. The following causes of solver resets are recorded:

- Zero Crossing
- Discrete Signal
- ZOH Signal
- Block Change
- Initial Reset
- Internal
See Also
solverprofiler.profileModel | Zero Crossing Explorer | States Explorer

Topics
“Examine Model Dynamics Using Solver Profiler”
“Solver Resets”
“Zero-Crossing Events”
“Solver Exception Events”
“Jacobian Logging and Analysis”

Introduced in R2016a
States Explorer

Visualize states in model

Description
Use the States Explorer to view individual state dynamics of your model

Open the States Explorer

- Profile the model using the Solver Profiler. Ensure that the States & Zero Crossing parameter is enabled.
- Select States in the Explore section of the Solver Profiler toolstrip.

See Also
solverprofiler.profileModel | Solver Profiler | Zero Crossing Explorer

Introduced in R2016a
Zero Crossing Explorer

Visualize and interactively explore zero-crossing events

Description
Use the Zero Crossing Explorer to visualize and analyze zero crossing events that occur for each state in your model.

Open the Zero Crossing Explorer

- Profile the model using the Solver Profiler. Ensure that the States & Zero Crossing parameter is enabled.
- Select Zero Crossing in the Explore section of the Solver Profiler toolstrip.

Examples

Interactively Explore Zero Crossings

Use the Zero Crossing Explorer to examine clusters of zero crossing events

Open the ex_spi_show_zc model and launch the Solver Profiler.

Enable the States & Zero Crossing parameter in the Log section of the toolstrip and run the Solver Profiler.

When profiling has completed, launch the Zero Crossing Explorer
There is an increase in the density of zero crossing events halfway through the simulation. Zoom in on the **event density** plot between the 5 and 6 second mark.
A closer examination reveals that each bounce corresponds to two zero crossings — from positive to negative and vice-versa. This is an artefact of numerically solving state dynamics that approach 0 value. Around 5.48 seconds, the ball being simulated in the model comes to 'rest', hence the isolated single zero crossing event.

- "Zero-Crossing Events"

See Also
Solver Profiler | States Explorer | solverprofiler.profileModel

Topics
"Zero-Crossing Events"
"Zero-Crossing Detection"

Introduced in R2019a
Objects — Alphabetical List
Simulink.Bus

Specify properties of buses

Description

Simulink.Bus objects, used with Simulink.BusElement objects, specify and validate the properties of a bus. When you simulate a model or update diagram, Simulink checks whether buses connected to blocks match specified Simulink.Bus objects.

You can use a Bus object to specify signal attributes. For example, you can specify signal attributes in Inport and Data Store Memory blocks.

To create and modify Bus objects in the base workspace or a data dictionary, you can use the Bus Editor or MATLAB commands. You cannot store Bus objects in model workspaces.

Creation

Syntax

busObject = Simulink.Bus

Description

busObject = Simulink.Bus returns a Bus object with these property values:

```
    Description: ''
    DataScope: 'Auto'
    HeaderFile: ''
    Alignment: -1
    Elements: [0×0 Simulink.BusElement]
```

The name of the Bus object is the name of the MATLAB variable to which you assign the Bus object.
Properties

**Description — Bus description**

character vector

Bus description, specified as a character vector. Use the description to document information about the Bus object, such as the kind of signal it applies to or where the Bus object is used. This information does not affect Simulink processing.

**Elements — Properties of signals within the bus**

array of Simulink.BusElement objects

Elements of the bus, specified as an array of Simulink.BusElement objects. Each BusElement object defines the name, data type, dimensions, and other properties of a signal within the bus.

**DataScope — Data type definition mode in generated code**

'Auto' (default) | 'Exported' | 'Imported'

Data type definition mode in generated code, specified as 'Auto', 'Exported', or 'Imported'. This property specifies whether during code generation the data type definition is imported from, or exported to, the header file specified with the HeaderFile property.

<table>
<thead>
<tr>
<th>Value</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>'Auto' (default)</td>
<td>Import the data type definition from the specified header file. If you do not specify the header file, export the data type definition to the default header file.</td>
</tr>
<tr>
<td>'Exported'</td>
<td>Export the data type definition to the specified header file or default header file.</td>
</tr>
<tr>
<td>'Imported'</td>
<td>Import the data type definition from the specified header file or default header file.</td>
</tr>
</tbody>
</table>

**HeaderFile — C header file used with data type definition**

character vector

C header file used with data type definition, specified as a character vector. Based on the value of the DataScope property, import the data type definition from or export the data type definition to the header file. The Simulink Coder software uses this property for code generation. Simulink software ignores this property.
By default, the generated `#include` directive uses the preprocessor delimiter " instead of `< and >. To generate the directive `#include <myTypes.h>`, specify `HeaderFile` as `<myTypes.h>`.

**Alignment — Data alignment boundary**

-1 (default) | integer

Data alignment boundary, specified as an integer, in number of bytes. The Simulink Coder software uses this property for code generation. Simulink software ignores this property.

The starting memory address for the data allocated for the bus is a multiple of the Alignment setting. If the object occurs in a context that requires alignment, you must specify an Alignment value with a positive integer that is a power of 2, not exceeding 128.

**Object Functions**

- `Simulink.Bus.cellToObject` Convert cell array containing bus information to bus objects
- `Simulink.Bus.createMATLABStruct` Create MATLAB structures using same hierarchy and attributes as bus signals
- `Simulink.Bus.createObject` Create bus objects from blocks or MATLAB structures
- `Simulink.Bus.objectToCell` Use bus objects to create cell array containing bus information
- `Simulink.Bus.save` Save bus objects in MATLAB file

**Examples**

**Create Bus Objects and BusElement Objects**

This example shows how to create a Bus object and its associated BusElement objects in the base workspace.

Create a Bus object named CONTROL.

```matlab
CONTROL = Simulink.Bus;
```

Create and specify properties for two BusElement objects, named VALVE1 and VALVE2.
valve1 = Simulink.BusElement;
valve1.Name = 'VALVE1';
valve1.Dimensions = 1;
valve1.DimensionsMode = 'Fixed';
valve1.DataType = 'double';
valve1.SampleTime = -1;
valve1.Complexity = 'real';

valve2 = Simulink.BusElement;
valve2.Name = 'VALVE2';
valve2.Dimensions = 1;
valve2.DimensionsMode = 'Fixed';
valve2.DataType = 'double';
valve2.SampleTime = -1;
valve2.Complexity = 'real';

Assign the elements to the CONTROL Bus object.

CONTROL.Elements = [valve1 valve2];

**Alternatives**

To interactively create a Bus object, use the **Bus Editor**.

To create Bus objects from blocks in a model, MATLAB data, and external C code, see “Create Bus Objects Programmatically”.

**See Also**

**Apps**
Bus Editor

**Objects**
Simulink.BusElement

**Topics**
“Specify Bus Properties with Bus Objects”
“Create Bus Objects Programmatically”
“Organize Data into Structures in Generated Code” (Simulink Coder)
“Data Alignment for Code Replacement” (Embedded Coder)

Introduced before R2006a
Simulink.BusElement

Specify properties of elements of buses

Description

A Simulink.BusElement object specifies the properties of a signal in a bus.

BusElement objects validate the properties of signals in a bus. A Simulink.Bus object contains BusElement objects. A BusElement object exists only within a Bus object. You can specify a bus object, but not a BusElement object, as a block parameter value. When you simulate a model or update diagram, Simulink checks whether the signals in a bus connected to blocks have the properties specified by the BusElement objects. If not, Simulink halts and displays an error message.

You can use the Simulink Bus Editor (see “Create and Specify Bus Objects”), or MATLAB commands (see “Create Bus Objects Programmatically”) to create and modify Bus objects and BusElement objects in the base MATLAB workspace.

Creation

Syntax

busElementObject = Simulink.BusElement

Description

busElementObject = Simulink.BusElement returns a BusElement object with these property values:

Name: 'a'
Complexity: 'real'
Dimensions: 1
DataType: 'double'
Min: []
Properties

**Name — Name of element**
character vector

Name of element, specified as a character vector.

**Complexity — Numeric type of element**
'real' (default) | 'complex'

Numeric type of the element, specified as 'real' or 'complex'.

**Dimensions — Dimensions of element**
array

Dimensions of element, specified as an array.

**DataType — Data type of element**
built-in Simulink data type | a Simulink.NumericType object | a Simulink.Bus object

Data type of element, specified as a built-in Simulink data type or Simulink.NumericType object. Examples of built-in data types include `double` and `uint8`. You can specify a Simulink.NumericType object whose `DataTypeMode` property is set to a value other than 'Fixed-point: unspecified scaling'. Specifying a Bus object allows you to create Bus objects that specify hierarchical buses (that is, buses that contain other buses).

**Min — Minimum value of element**
double | []

Minimum value of the element, specified as a double. This value must be a finite real double scalar or, if the element is a bus, the value must be empty, [].

**Max — Maximum value of element**
double | []
Maximum value of the element, specified as a double. This value must be a finite real double scalar or, if the element is a bus, the value must be empty, [ ].

**DimensionsMode — Specify how to handle size of element**

'Fixed' (default) | 'Variable'

Specify how to handle size of element, specified as 'Fixed' or 'Variable'.

**SampleTime — Sample time of element**

-1 (default) | double

Sample time of element, specified as a double. The sample time is the size of the interval between times when this signal value must be recomputed. If these conditions apply, use the default value of -1:

- The element is a bus.
- The bus that includes this element passes through a block that changes the bus sample time, such as a Rate Transition block.

**Unit — Physical unit for expressing element**

character vector

Physical unit for expressing element, specified as a character vector (for example, 'inches').

**Examples**

**Create Bus Objects and BusElement Objects**

This example shows how to create a Bus object and its associated BusElement objects in the base workspace.

Create a Bus object named CONTROL.

```matlab
CONTROL = Simulink.Bus;
```

Create and specify properties for two BusElement objects, named VALVE1 and VALVE2.

```matlab
valve1 = Simulink.BusElement;
valve1.Name = 'VALVE1';
```
valve1.Dimensions = 1;
valve1.DimensionsMode = 'Fixed';
valve1.DataType = 'double';
valve1.SampleTime = -1;
valve1.Complexity = 'real';

valve2 = Simulink.BusElement;
valve2.Name = 'VALVE2';
valve2.Dimensions = 1;
valve2.DimensionsMode = 'Fixed';
valve2.DataType = 'double';
valve2.SampleTime = -1;
valve2.Complexity = 'real';

Assign the elements to the CONTROL Bus object.

CONTROL.Elements = [valve1 valve2];

**Alternatives**

To interactively create a BusElement object, use the **Bus Editor**.

To create BusElement objects from blocks in a model, MATLAB data, and external C code, see “Create Bus Objects Programmatically”.

**Compatibility Considerations**

**Simulink.BusElement objects will no longer support the SampleTime property**

*Not recommended starting in R2019b*

In a future release, the SampleTime property of Simulink.BusElement objects will be removed.

To prepare:

- When defining BusElement objects, use the default SampleTime value (-1).
To specify the sample time for an element of a bus, use the `SampleTime` block parameter of corresponding blocks. For example, you can use In Bus Element, Out Bus Element, and Signal Specification blocks to specify sample time.

**Simulink.BusElement objects will no longer support the `SamplingMode` property**

*Not recommended starting in R2016b*

In R2016b, the `SamplingMode` property of `Simulink.BusElement` objects was removed. Scripts that use the `SamplingMode` property of `Simulink.BusElement` objects continue to work. `Simulink.Bus.cellToObject` continues to require the `SamplingMode` field and `Simulink.Bus.objectToCell` continues to include the sampling mode in the output cell arrays.

In a future release, support for the `SamplingMode` property will be removed.

To specify whether a signal is sample-based or frame-based, define the sampling mode of input signals at the block level instead of at the signal level.

**See Also**

`Simulink.Bus`

**Topics**

“Create Bus Objects Programmatically”
“Save Bus Objects”
“Signal Names and Labels”
“Specify Sample Time”
“Variable-Size Signal Basics”

**Introduced before R2006a**
**ConfigSet**

Model configuration set

**Description**

Use the `ConfigSet` object to access a model configuration set. Get information about your configuration set, manage configuration parameters, and manage components.

**Creation**

Use the `getActiveConfigSet` function to get the active configuration set for a model. Use the `getConfigSet` function to get a model configuration set by name.

**Properties**

**Components — Components of the configuration set**

cell array of `Simulink.ConfigComponent` objects

Components of the configuration set, specified by a cell array of `Simulink.ConfigComponent` objects.

**Description — Description of the configuration set**

character vector

Description of the configuration set, specified as a character vector. Use the description to provide additional information about a configuration set, such as its purpose.

**Name — Name of the configuration set**

character vector

Name of the configuration set, specified by a character vector. This name represents the configuration set in the Model Explorer.
Object Functions

- attachComponent: Attach a component to a configuration set
- copy: Copy a configuration set
- getComponent: Get a configuration set component
- getFullName: Get parameter names and values
- getModel: Get parameter names and values
- get_param: Get parameter names and values
- isActive: Get parameter names and values
- isValidParam: Get parameter names and values
- saveAs: Set system and block parameter values
- setPropEnabled: Set system and block parameter values
- set_param: Set system and block parameter values

Examples

Get the Active Configuration Set for a Model

This example gets the active configuration set of the currently selected model.

```matlab
hCs = getActiveConfigSet(gcs);
```

See Also

- getActiveConfigSet | getConfigSet | getConfigSets

Topics

- “About Configuration Sets”
- “Manage a Configuration Set”

Introduced in R2006a
Simulink.HMI.InstrumentedSignals

Save and restore signal logging specification

Description

A Simulink.HMI.InstrumentedSignals object acts as a specification for logged signals in a model, including signals in subsystems, library instances, and Stateflow charts. You can get a Simulink.HMI.InstrumentedSignals object for the logging configuration in your model using the get_param function with the InstrumentedSignals model parameter. You can save several logging configurations as Simulink.HMI.InstrumentedSignals objects and easily swap the logging configuration for your model using set_param.

Note An InstrumentedSignals object for a model does not include signals logged inside referenced models. To get a list of logged signals in a referenced model, create an InstrumentedSignals object for the referenced model.

Creation

The get_param function returns a Simulink.HMI.InstrumentedSignals object for the InstrumentedSignals model parameter.

Properties

Model — Model name
character vector

Name of the model that corresponds to the list of logged signals.
Example: 'sldemo_fuelsys'

Count — Number of logged signals
integer
Number of logged signals in the model.
Example: 10
Data Types: uint32

Examples

Save and Restore a Set of Logged Signals

This example shows how to use the Simulink.HMI.InstrumentedSignals object to save a set of logged signals to restore after running a simulation with a different signal logging configuration.

Save the Initial Signal Logging Configuration

This example uses the sldemo_fuelsys model, which is configured to log 10 signals. Open the model and use the get_param function to get a Simulink.HMI.InstrumentedSignals object representing the signal logging configuration.

load_system sldemo_fuelsys
initSigs = get_param('sldemo_fuelsys','InstrumentedSignals');

You can save the initial signal logging configuration in a MAT-file for later use.

save initial_instSigs.mat initSigs

Remove All Logging Badges

To return to a baseline of no logged signals, you can use the set_param function to remove all logging badges from signals in your model. Then, you can easily select a different configuration of signals to log in the Simulink™ Editor or using the Simulink.sdi.markSignalForStreaming function.

set_param('sldemo_fuelsys','InstrumentedSignals',[])

Restore Saved Logging Configuration

After working with a different set of logged signals, you can restore a saved configuration using the Simulink.HMI.InstrumentedSignals object. For example, if you saved the
logging configuration to a MAT-file, you can load the MAT-file contents into the workspace and use the set_param function to restore the previously saved logging configuration.

load initial_instSigs.mat

set_param('sldemo_fuelsys','InstrumentedSignals',initSigs)

**See Also**

Simulink.HMI.SignalSpecification | Simulink.sdi.markSignalForStreaming

**Topics**

“View Data with the Simulation Data Inspector”
“Export Signal Data Using Signal Logging”
“Configure a Signal for Logging”

**Introduced in R2015b**
Simulink.HMI.ParamSourceInfo

Information about Dashboard block variable and parameter connections

Description

Use a Simulink.HMI.ParamSourceInfo object to connect a Dashboard block to a variable or parameter programmatically using the set_param function. The get_param function returns a Simulink.HMI.ParamSourceInfo object when you use the function to query the connection information for a Dashboard block connected to a variable or parameter.

The Simulink.HMI.ParamSourceInfo object contains four properties. Some of the properties apply to connecting Dashboard blocks to parameters, and some apply for connecting Dashboard blocks to variables. Not all fields have a value for a connection because a given Dashboard block connects to either a parameter or a variable.

Creation

Syntax

paramSourceInfo = Simulink.HMI.ParamSourceInfo

Description

paramSourceInfo = Simulink.HMI.ParamSourceInfo creates the empty Simulink.HMI.ParamSourceInfo object, paramSourceInfo.

Properties

BlockPath — Path to the block associated with the parameter or variable
Simulink.BlockPath
Simulink.BlockPath object for the block associated with the parameter or variable. You can create a Simulink.BlockPath object for a block by passing a character array describing the path to Simulink.BlockPath. You can get a character array describing the block path for a selected block using the gcb function.

Example: Simulink.BlockPath('vdp/Mu')

**ParamName — Name of the connected tunable block parameter**
character array

Name of the connected tunable block parameter, specified as a character array. A Simulink.HMI.ParamSourceInfo object connected to a variable does not have a value for the ParamName property.

Example: Gain

**VarName — Name of the connected variable**
character array

Name of the connected variable, specified as a character array. A Simulink.HMI.ParamSourceInfo object connected to a tunable parameter does not have a value for the VarName property.

Example: Mu

**WksType — Source workspace for the connected variable**
'base' | 'model' | data dictionary file path

Source workspace for the connected variable, specified as a character array. The source workspace can be the base workspace, model workspace, or a data dictionary. When the source workspace is a data dictionary, the WksType property value is the file path for the data dictionary, specified as a character array. A Simulink.HMI.ParamSourceInfo object connected to a tunable parameter does not have a value for the WksType property.

Example: 'modelData.sldd'

**Examples**
**Programmatically Add and Connect a Dashboard Block**

This example shows how to programmatically add Dashboard blocks to a model and connect them to elements in the model. The example adds a Dashboard Scope block and a Slider block to the `vdp` model and connects and configures the blocks.

**Add Blocks**

Use the `add_block` function to add a Dashboard Scope block and a Slider block to the `vdp` model. This example also specifies the position of the blocks.

```matlab
open_system('vdp')
scopePos = [550 140 800 305];
sliderPos = [550 30 800 120];
add_block('simulink_hmi_blocks/Dashboard Scope','vdp/Dashboard Scope','Position',scopePos)
add_block('simulink_hmi_blocks/Slider','vdp/Slider','Position',sliderPos)
```

**Connect the Dashboard Scope Block**

Use a cell array of `Simulink.HMI.SignalSpecification` objects to connect the Dashboard block to the `x1` and `x2` signals. The `Simulink.HMI.SignalSpecification` object specifies a connected signal using the block path and port index for the source of the signal. The default value for the `OutputPortIndex` is 1, so this example does not specify the output port for the `x1` and `x2` signals.

```matlab
x1_sigSpec = Simulink.HMI.SignalSpecification;
x1_sigSpec.BlockPath = Simulink.BlockPath('vdp/x1');

x2_sigSpec = Simulink.HMI.SignalSpecification;
x2_sigSpec.BlockPath = Simulink.BlockPath('vdp/x2');

connection_dashboardScope = {x1_sigSpec x2_sigSpec};
```

Use the `set_param` function to configure the connected signals for the Dashboard Scope block.

```matlab
set_param('vdp/Dashboard Scope','Binding',connection_dashboardScope)
```

**Connect and Configure the Slider Block**

Use a `Simulink.HMI.ParamSourceInfo` object and the `set_param` function to connect the Slider block to the Gain parameter of the Mu block. To connect a parameter, the
Simulink.HMI.ParamSourceInfo needs to specify the block path for the block that corresponds to the parameter and the name of the parameter.

```matlab
slider_param = Simulink.HMI.ParamSourceInfo;
slider_param.BlockPath = Simulink.BlockPath('vdp/Mu');
slider_param.ParamName = 'Gain';

set_param('vdp/Slider','Binding',slider_param)
```

Configure the scale for the slider for a range of 1 to 10 with a tick mark spacing of 1.

```matlab
slider_limits = [1 1 10];
set_param('vdp/Slider','Limits',slider_limits)
```

See Also

Check Box | Combo Box | Edit | Knob | Push Button | Radio Button | Rocker Switch | Rotary Switch | Simulink.HMI.SignalSpecification | Slider | Slider Switch | Toggle Switch

Introduced in R2019a
Simulink.HMI.SignalSpecification

Programmatically connect a Dashboard block to a signal

Description

Use a Simulink.HMI.SignalSpecification object to programmatically connect a Dashboard block to a signal.

The SignalSpecification object contains the block path, port index, and frame processing mode for a signal. To connect a Dashboard block to a signal, specify the corresponding SignalSpecification object as the value for the Binding parameter for the block using the set_param function.

Creation

Syntax

sigSpec = Simulink.HMI.SignalSpecification

Description

sigSpec = Simulink.HMI.SignalSpecification creates an empty Simulink.HMI.SignalSpecification object. Specify the block path and port index that correspond to the origin of the signal you want to connect.

Properties

BlockPath — Path of the block with the signal as an output
Simulink.BlockPath (default) | character vector

Block path for the block that has the signal you want to connect as an output, specified as a character vector or a Simulink.BlockPath object.
Example: `sigSpec.BlockPath = 'vdp/Mu';`
Example: `sigSpec.BlockPath = myBlockPath;`

**OutputPortIndex** — Block output port corresponding to the signal

1 (default) | scalar

Output port index corresponding to the signal, specified as a scalar, real integer.
Example: `sigSpec.OutputPortIndex = 2;`

**FrameProcessingMode** — Processing mode for the signal data

"sample" (default) | "frame"

Processing mode for the signal data, specified as sample or frame.
- **sample** — Each element in a sample of the signal is processed as a channel.
- **frame** — Each column in a sample of the signal is processed as a channel.

Example: `sigSpec.FrameProcessingMode = "frame";`

**Note** Only the Dashboard Scope block supports frame-based data.

---

**Examples**

**Programmatically Add and Connect a Dashboard Block**

This example shows how to programmatically add Dashboard blocks to a model and connect them to elements in the model. The example adds a Dashboard Scope block and a Slider block to the `vdp` model and connects and configures the blocks.

**Add Blocks**

Use the `add_block` function to add a Dashboard Scope block and a Slider block to the `vdp` model. This example also specifies the position of the blocks.

```matlab
open_system('vdp')
scopePos = [550 140 800 305];
sliderPos = [550 30 800 120];
```
Connect the Dashboard Scope Block

Use a cell array of Simulink.HMI.SignalSpecification objects to connect the Dashboard block to the x1 and x2 signals. The Simulink.HMI.SignalSpecification object specifies a connected signal using the block path and port index for the source of the signal. The default value for the OutputPortIndex is 1, so this example does not specify the output port for the x1 and x2 signals.

```matlab
x1_sigSpec = Simulink.HMI.SignalSpecification;
x1_sigSpec.BlockPath = Simulink.BlockPath('vdp/x1');

x2_sigSpec = Simulink.HMI.SignalSpecification;
x2_sigSpec.BlockPath = Simulink.BlockPath('vdp/x2');

connection_dashboardScope = {x1_sigSpec x2_sigSpec};
```

Use the set_param function to configure the connected signals for the Dashboard Scope block.

```matlab
set_param('vdp/Dashboard Scope','Binding',connection_dashboardScope)
```

Connect and Configure the Slider Block

Use a Simulink.HMI.ParamSourceInfo object and the set_param function to connect the Slider block to the Gain parameter of the Mu block. To connect a parameter, the Simulink.HMI.ParamSourceInfo needs to specify the block path for the block that corresponds to the parameter and the name of the parameter.

```matlab
slider_param = Simulink.HMI.ParamSourceInfo;
slider_param.BlockPath = Simulink.BlockPath('vdp/Mu');
slider_param.ParamName = 'Gain';

set_param('vdp/Slider','Binding',slider_param)
```

Configure the scale for the slider for a range of 1 to 10 with a tick mark spacing of 1.

```matlab
slider_limits = [1 1 10];
set_param('vdp/Slider','Limits',slider_limits)
```
See Also
Simulink.BlockPath | get_param | set_param

Topics
“View Data with the Simulation Data Inspector”
“Tune and Visualize Your Model with Dashboard Blocks”

Introduced in R2015b
MATLABFunctionConfiguration

Configure MATLAB Function block properties programmatically

Description

The MATLABFunctionConfiguration object controls the MATLAB Function block properties such as the description, sample time, and the function script. You can create a configuration object using `get_param` and change the property values of the object using dot notation.

Creation

`myMFBConfigobj = get_param(gcb, 'MATLABFunctionConfiguration')` creates a MATLAB Function block configuration object for the selected MATLAB Function block.

Properties

Path — Path of the block relative to the model
character vector (default)

Display the path of the MATLAB Function block relative to the model.

Data Types: char

FunctionScript — Block function script
character vector (default)

Display the script for the MATLAB Function block.

Data Types: char

UpdateMethod — Block update method
Inherited (default) | Discrete | Continuous

Update method of the block, specified as an enumerated string.
Specify an `UpdateMethod` using one of these values:

- **Inherited** — Input to the MATLAB Function block determines when the block is updated during a simulation. The input is explicitly triggered by a signal on its trigger port originated from a connected Simulink block. If the `UpdateMethod` is changed to the 'Inherited', the sample time automatically sets to -1.

- **Discrete** — Simulink model generates an implicit event at regular time intervals to awaken the block at the rate you specify in the `SampleTime` property. Other blocks in the model can have different sample times.

- **Continuous** — The MATLAB Function block updates at major time steps only, although it computes outputs and local continuous variables during minor and major time steps. If the `UpdateMethod` is changed to 'Continuous', the sample time automatically sets to 0.

When you set the `UpdateMethod` to Inherited or Continuous and attempt to change `SampleTime`, Simulink displays a warning and ignores the input sample time.

**UI Use**

Open the **Ports and Data Manager** and change the **Update Method** to the method you choose.

Data Types: enumerated

**SampleTime — Block sample time**

character vector (default) | '0' | '-1'

Change the sample time of the MATLAB Function block. Sample time is only valid when the `UpdateMethod` is set to Discrete. If the `UpdateMethod` is set to 'Continuous' or 'Inherited', any changes made to the sample time are ignored.

**UI Use**

Open the **Ports and Data Manager** and change the **Sample Time**.

Data Types: char

**Description — Block Description**

character vector (default)

Description of the block, specified as a character vector.
**UI Use**

Open the **Ports and Data Manager** and change the **Description** to the text you choose.

Data Types: char

**DocumentLink — Documentation link for the block**
character vector (default)

Documentation link for the block, specified as a character vector.

**UI Use**

Open the **Ports and Data Manager** and change the **Document Link** to your block documentation link.

Data Types: char

**SupportVariableSizing — Support variable input and output data**
1 (default) | 0 | true | false

Specify whether the simulation supports variable input and output data dimensions.

**UI Use**

Open the **Ports and Data Manager** and check the **Support variable-size arrays** to turn this setting on.

Data Types: logical

**AllowDirectFeedthrough — Direct feedthrough status of the block**
1 (default) | 0 | true | false

Display or change whether the block has direct feedthrough. The block is considered to have direct feedthrough if the output of the block directly depends on the input of the block. The block does not have direct feedthrough if the outputs of the block depend on the internal states and properties rather than the input of the block.

**UI Use**

Open the **Ports and Data Manager** and check the **Allow direct feedthrough** to turn this setting on.

Data Types: logical
**Saturate on Integer Overflow — Specify whether overflows saturate**

1 (default) | 0 | true | false

Specify whether the integer overflow saturates. If this setting is set to `true`, the overflows saturate to either the minimum or maximum value that the data type can be present. If the setting is `false`, the overflow wraps to the appropriate value that the data type can represent. If your model has a possible overflow, and you want to apply saturation protection in your generated code, it is recommended to have this setting on. If you have performance restrictions, and not concerned with the integer overflow, you can set this property to `false`.

**UI Use**

Open the **Ports and Data Manager** and check the **Saturate on integer overflow** box to turn this setting on.

Data Types: **logical**

**TreatAsFi — Specify data types as fi objects**

'FixedPoint' (default) | 'FixedPointAndInteger'

Specify which data types to treat as `fi` data types. To learn more about `fi` objects, see “`fi` Object Properties” (Fixed-Point Designer).

**UI Use**

Open the **Ports and Data Manager** and select an option from the **Treat these inherit Simulink signal types as fi objects** menu.

Data Types: **string**

**FimathMode — Specify fixed-point settings**

'SameAsMATLAB' (default) | 'UserSpecified'

Specify which fixed-point math settings to use. You can use the current MATLAB fixed-point math settings, or you can enable the edit box to specify the desired fixed-point math settings. To learn more about fixed-point math, see `fimath`.

**UI Use**

Open the **Ports and Data Manager** and choose from the **MATLAB Function fimath** settings.

Data Types: **string**
Fimath — Specify the fixed-point math settings

`fimath('SumMode','FullPrecision')` (default) | `fimath(...'PropertyName',PropertyValue...)`

Specify the fimath construction function you want to use.
`fimath(...'PropertyName',PropertyValue...)` allows you to set the attributes of a fimath object using property name/property value pairs. To learn more fimath object properties, see “fimath Object Properties” (Fixed-Point Designer).

**UI Use**

Open the Ports and Data Manager and edit the MATLAB Function fimath box with the appropriate fimath expression.

Data Types: string

**Examples**

**Create a MATLAB Function Block Configuration Object**

This example creates a MATLAB Function block configuration object using the 'sldemo_radar_eml' model.

```matlab
open_system('sldemo_radar_eml')
myMFBConfig = get_param('sldemo_radar_eml/MATLAB Function','MATLABFunctionConfiguration');
myMFBConfig.Description = 'Radar data processing tracker using Extended Kalman';
```

**See Also**

MATLAB Function

**Topics**

“Radar Tracking Using MATLAB Function Block” on page 15-297
“Create Custom Functionality Using MATLAB Function Block”

Introduced in R2019b
FunctionPortSpecification

Query and configure C Caller block properties

Description

Query and configure the C Caller block function definition and return argument. Configure the C Caller block input arguments programmatically.

Creation

myCCallerConfigObj = get_param(gcb, 'FunctionPortSpecification')
creates a C Caller block configuration object for the selected C Caller block.

Properties

CPrototype — C function mapping to the C Caller block
character vector
Displays the mapping of the C function input to the C Caller block in the model. You cannot edit this property using the command line. To edit the C function mapping, edit the source code of the C Caller block.

Example: real_T add(real_T u1, real_T u2);
Data Types: char

InputArguments — List of function input arguments
FunctionArgument object
Returns the list of function input arguments in a “FunctionArgument Object” on page 8-31.

ReturnArgument — List of function output arguments
FunctionArgument scalar
Returns a scalar or an empty “FunctionArgument Object” on page 8-31.
Data Types: `char`

### FunctionArgument Object

The `FunctionArgument` is a class that carries port specification information of a C Caller block programmatically. The `FunctionArgument` object has these properties:

1. **Name** — Name of the variable in the source code, specified as a character vector. This property is read-only.
2. **PortNumber** — Port number of the `InputArgument` that has been edited, specified as `uint32`. This property is read-only.
3. **Size** — Size of the Simulink port dimensions specified as a character vector. You can edit this property if it is allowed in the function definition.
4. **Type** — Simulink data type for the specified port specified as a character vector. You can edit this property if it is allowed in the function definition.
5. **Label** — Port label on the Simulink block specified as a character vector. You can change the port label programmatically, for example:

   ```matlab
   myCCallerObj.my_CCaller.InputArguments(1).Label = 'inputport1'
   my_CCallerObj =
   FunctionPortSpecification with properties:
   CPrototype: 'real_T add(real_T u1, real_T u2);'
   InputArguments: [1×2 Simulink.CustomCode.FunctionArgument]
   ReturnArgument: [1×1 Simulink.CustomCode.FunctionArgument]
   ``

6. **Scope** — Mapping of the Simulink scope to the input argument specified as a character vector. If the variable is defined as a constant qualifier, the argument can be an input or a parameter. If no constant qualifier exists, you can change an output argument to an `input`, `inputoutput`, or to a `parameter`. If your scope fits any of the scenarios above, you can change it programmatically:

   ```matlab
   my_CCallerObj.InputArguments(1).Scope = 'parameter'
   my_CCallerObj =
   FunctionPortSpecification with properties:
   CPrototype: 'extern real_T add(real_T u1, real_T u2);'
   ```
Examples

Create a C Caller Configuration Object

This example creates a C Caller configuration object using the 'slexCCallerExample' demo model.

```matlab
my_CCallerObj = get_param('slexCCallerExample/C Caller','FunctionPortSpecification')
my_CCallerObj =
    FunctionPortSpecification with properties:
        CPrototype: 'real_T add(real_T u1, real_T u2);'
        InputArguments: [1×2 Simulink.CustomCode.FunctionArgument]
        ReturnArgument: [1×1 Simulink.CustomCode.FunctionArgument]
```

Create a FunctionArgument object for the 1st input port:

```matlab
inargone = my_CCallerObj.InputArguments(1)
inargone =
    FunctionArgument with properties:
        Name: 'u1'
        PortNumber: 1
        Size: '1'
        Type: 'double'
        Label: 'u1'
        Scope: 'Parameter'
```

Change the label for the first input port:

```matlab
inargone.Label = 'inputport1'
inargone =
    FunctionArgument with properties:
```
Name: 'ul'
PortNumber: 1
  Size: '1'
  Type: 'double'
Label: 'inputport1'
Scope: 'Parameter'

See Also
C Caller | MATLABFunctionConfiguration

Topics
“Call C Functions Using C Caller Block” on page 15-699
“Integrate C Code Using C Caller Blocks”

Introduced in R2019b
Fixed-Point Tool

- “Fixed-Point Tool Parameters and Dialog Box” on page 9-2
- “Advanced Settings” on page 9-20
Fixed-Point Tool Parameters and Dialog Box

The Fixed-Point Tool includes the following components:

- **Main toolbar**
- **Model Hierarchy pane**
- **Contents pane**
- **Dialog pane**

Main Toolbar

The Fixed-Point Tool's main toolbar appears near the top of the Fixed-Point Tool window under the Fixed-Point Tool's menu.

The toolbar contains the following buttons that execute commonly used Fixed-Point Tool commands:

<table>
<thead>
<tr>
<th>Button</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>![Open Advisor]</td>
<td>Open the Fixed-Point Advisor to prepare the model for conversion to fixed point.</td>
</tr>
<tr>
<td>![Simulate]</td>
<td>Simulate a model and store the run results.</td>
</tr>
<tr>
<td>![Pause]</td>
<td>Pause a simulation.</td>
</tr>
<tr>
<td>![Stop]</td>
<td>Stop a simulation.</td>
</tr>
<tr>
<td>![Analyze]</td>
<td>Analyze model and store derived minimum and maximum results.</td>
</tr>
<tr>
<td>![Propose DT]</td>
<td>Propose data types. Propose fraction lengths for specified word lengths or propose word lengths for specified fraction lengths.</td>
</tr>
<tr>
<td>![Apply]</td>
<td>Apply accepted data types.</td>
</tr>
<tr>
<td>![Compare]</td>
<td>Compare selected runs.</td>
</tr>
<tr>
<td>![Difference]</td>
<td>Create a difference plot for the selected signals.</td>
</tr>
<tr>
<td>![Plot]</td>
<td>Plot the selected signal.</td>
</tr>
</tbody>
</table>
Button | Usage
--- | ---
| | Create a histogram plot for the selected signal.

The toolbar also contains the **Show** option:

![Show Option](image)

The **Show** option specifies the type of results to display in the **Contents** pane. The **Contents** pane displays information only after you simulate a system or propose fraction lengths. If there are no results that satisfy a particular filter option, the list will be blank.

<table>
<thead>
<tr>
<th>Show Option</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>All results</td>
<td>Displays all results for the selected tree node.</td>
</tr>
<tr>
<td><strong>Signal Logging results</strong></td>
<td>For the selected tree node, displays blocks whose output ports have logged signal data. The Fixed-Point tool marks these blocks with the logged signal icon [log]. Note: You can plot simulation results associated with logged signal data using the Simulation Data Inspector.</td>
</tr>
<tr>
<td><strong>Min/Max results</strong></td>
<td>For the selected tree node, displays blocks that record design Min/Max, simulation Min/Max, and overflow data. <strong>Prerequisites:</strong> Fixed-point instrumentation mode should not be set to Force Off.</td>
</tr>
<tr>
<td><strong>Overflows</strong></td>
<td>For the selected tree node, displays blocks that have non-zero overflows recorded. If a block has its Saturate on integer overflow option selected, overflow information appears in the Satuations column, otherwise it appears in the OverflowWraps column.</td>
</tr>
<tr>
<td><strong>Show Option</strong></td>
<td><strong>Result</strong></td>
</tr>
<tr>
<td>-----------------------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>Conflicts with proposed data types</strong></td>
<td>For the selected tree node, displays results that have potential data typing or scaling issues.</td>
</tr>
<tr>
<td><strong>Prerequisites:</strong> This information is available only after you propose data types.</td>
<td></td>
</tr>
<tr>
<td>The Fixed-Point Tool marks these results with a yellow or red icon, as shown here:</td>
<td></td>
</tr>
<tr>
<td><img src="image" alt="Icon" /> The proposed data type poses potential issues for this object. Open the Result Details tab to review these issues.</td>
<td></td>
</tr>
<tr>
<td><img src="image" alt="Icon" /> The proposed data type will introduce errors if applied to this object. Open the Result Details tab for details about how to resolve these issues.</td>
<td></td>
</tr>
<tr>
<td><strong>Groups that must share the same data type</strong></td>
<td>For the selected tree node, displays blocks that must share the same data type because of data type propagation rules.</td>
</tr>
<tr>
<td><strong>Prerequisites:</strong> This information is available only after you propose fraction lengths.</td>
<td></td>
</tr>
<tr>
<td>The Fixed-Point Tool allocates an identification tag to blocks that must share the same data type. This identification tag is displayed in the <strong>DTGroup</strong> column as follows:</td>
<td></td>
</tr>
<tr>
<td>• If the selected tree node is the model root</td>
<td>All results for the model are listed. The <strong>DTGroup</strong> column is sorted by default so that you can easily view all blocks in a group.</td>
</tr>
<tr>
<td>• If the selected tree node is a subsystem</td>
<td>The identification tags have a suffix that indicates the total number of results in each group. For example, <strong>G2(2)</strong> means group G2 has 2 members. This information enables you to see how many members of a group belong to the selected subsystem and which groups share data types across subsystem boundaries.</td>
</tr>
</tbody>
</table>
Model Hierarchy Pane

The Model Hierarchy pane displays a tree-structured view of the Simulink model hierarchy. The first node in the pane represents a Simulink model. Expanding the root node displays subnodes that represent the model's subsystems, MATLAB Function blocks, Stateflow charts, and referenced models.

The Fixed-Point Tool's Contents pane displays elements that comprise the object selected in the Model Hierarchy pane. The Dialog pane provides parameters for specifying the selected object's data type override and fixed-point instrumentation mode. You can also specify an object's data type override and fixed-point instrumentation mode by right-clicking on the object. The Model Hierarchy pane indicates the value of these parameters by displaying the following abbreviations next to the object name:

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Parameter Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fixed-point instrumentation mode</td>
<td></td>
</tr>
<tr>
<td>mmo</td>
<td>Minimums, maximums and overflows</td>
</tr>
<tr>
<td>o</td>
<td>Overflows only</td>
</tr>
<tr>
<td>fo</td>
<td>Force off</td>
</tr>
<tr>
<td>Data type override</td>
<td></td>
</tr>
<tr>
<td>scl</td>
<td>Scaled double</td>
</tr>
<tr>
<td>dbl</td>
<td>Double</td>
</tr>
<tr>
<td>sgl</td>
<td>Single</td>
</tr>
<tr>
<td>off</td>
<td>Off</td>
</tr>
</tbody>
</table>

Contents Pane

The Contents pane displays a tabular view of objects that log fixed-point data in the system or subsystem selected in the Model Hierarchy pane. The table rows correspond to model objects, such as blocks, block parameters, and Stateflow data. The table columns correspond to attributes of those objects, such as the data type, design minimum and maximum values, and simulation minimum and maximum values.

The Contents pane displays information only after you simulate a system, analyze the model to derive minimum and maximum values, or propose fraction lengths.
You can control which of the following columns the Fixed-Point Tool displays in this pane. For more information, see “Customizing the Contents Pane View” on page 9-8.

<table>
<thead>
<tr>
<th>Column Label</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accept</td>
<td>Check box that enables you to selectively accept the Fixed-Point Tool's data type proposal.</td>
</tr>
<tr>
<td>CompiledDesignMax</td>
<td>Compile-time information for DesignMax.</td>
</tr>
<tr>
<td>CompiledDesignMin</td>
<td>Compile-time information for DesignMin.</td>
</tr>
<tr>
<td>CompiledDT</td>
<td>Compile-time data type. This data type appears on the signal line in $sfix$ format. See “Fixed-Point Data Type and Scaling Notation” (Fixed-Point Designer).</td>
</tr>
<tr>
<td>DerivedMax</td>
<td>Maximum value the Fixed-Point tool derives for this signal from design ranges specified for blocks.</td>
</tr>
<tr>
<td>DerivedMin</td>
<td>Minimum value the Fixed-Point tool derives for this signal from design ranges specified for blocks.</td>
</tr>
<tr>
<td>DesignMax</td>
<td>Maximum value the block specifies in its parameter dialog box, for example, the value of its <strong>Output maximum</strong> parameter.</td>
</tr>
<tr>
<td>DesignMin</td>
<td>Minimum value the block specifies in its parameter dialog box, for example, the value of its <strong>Output minimum</strong> parameter.</td>
</tr>
<tr>
<td>DivByZero</td>
<td>Number of divide-by-zero instances that occur during simulation.</td>
</tr>
<tr>
<td>DTGroup</td>
<td>Identification tag associated with objects that share data types.</td>
</tr>
<tr>
<td>InitValueMax</td>
<td>Maximum initial value for a signal or parameter. Some model objects provide parameters that allow you to specify the initial values of their signals. For example, the Constant block includes a <strong>Constant value</strong> that initializes the block output signal.</td>
</tr>
</tbody>
</table>

**Note** The Fixed-Point Tool uses this parameter when it proposes data types.
<table>
<thead>
<tr>
<th>Column Label</th>
<th>Description</th>
</tr>
</thead>
</table>
| **InitValueMin** | Minimum initial value for a signal or parameter. Some model objects provide parameters that allow you to specify the initial values of their signals. For example, the Constant block includes a **Constant value** that initializes the block output signal.  

**Note** The Fixed-Point Tool uses this parameter when it proposes data types. |
| **LogSignal** | Check box that allows you to enable or disable signal logging for an object. |
| **ModelRequiredMin** | Minimum value of a parameter used during simulation. For example, the n-D Lookup Table block uses the **Breakpoints** and **Table data** parameters to perform its lookup operation and generate output. In this example, the block uses more than one parameter so the Fixed-Point Tool sets **ModelRequiredMin** to the minimum of the minimum values of all these parameters.  

**Note** The Fixed-Point Tool uses this parameter when it proposes data types. |
| **ModelRequiredMax** | Maximum value of a parameter used during simulation. For example, the n-D Lookup Table block uses the **Breakpoints** and **Table data** parameters to perform its lookup operation and generate output. In this example, the block uses more than one parameter so the Fixed-Point Tool sets **ModelRequiredMax** to the maximum of the maximum values of all these parameters.  

**Note** The Fixed-Point Tool uses this parameter when it proposes data types. |
| **Name** | Identifies path and name of block. |
| **OverflowWraps** | Number of overflows that wrap during simulation. |
| **ProposedDT** | Data type that the Fixed-Point Tool proposes. |
### Column Label

<table>
<thead>
<tr>
<th>Column Label</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ProposedMax</td>
<td>Maximum value that results from the data type the Fixed-Point Tool proposes.</td>
</tr>
<tr>
<td>ProposedMin</td>
<td>Minimum value that results from the data type the Fixed-Point Tool proposes.</td>
</tr>
<tr>
<td>Run</td>
<td>Indicates the run name for these results.</td>
</tr>
<tr>
<td>Saturations</td>
<td>Number of overflows that saturate during simulation.</td>
</tr>
<tr>
<td>SimDT</td>
<td>Data type the block uses during simulation. This data type appears on the signal line in <code>sfix</code> format. See “Fixed-Point Data Type and Scaling Notation” (Fixed-Point Designer).</td>
</tr>
<tr>
<td>SimMax</td>
<td>Maximum value that occurs during simulation.</td>
</tr>
<tr>
<td>SimMin</td>
<td>Minimum value that occurs during simulation.</td>
</tr>
<tr>
<td>SpecifiedDT</td>
<td>Data type the block specifies in its parameter dialog box, for example, the value of its <strong>Output data type</strong> parameter.</td>
</tr>
</tbody>
</table>

### Customizing the Contents Pane View

You can customize the **Contents** pane in the following ways:

- “Using Column Views” on page 9-8
- “Changing Column Order and Width” on page 9-10
- “Sorting by Columns” on page 9-10

### Using Column Views

The Fixed-Point Tool provides the following standard Column Views:

<table>
<thead>
<tr>
<th>View Name</th>
<th>Columns Provided</th>
<th>When Does the Fixed-Point Tool Display this View?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simulation View</td>
<td><strong>Name, Run, CompiledDT, SpecifiedDT, SimMin, SimMax, DesignMin, DesignMax, OverflowWraps, Saturations</strong></td>
<td>After a simulating minimum and maximum values.</td>
</tr>
<tr>
<td>View Name</td>
<td>Columns Provided</td>
<td>When Does the Fixed-Point Tool Display this View?</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td>-----------------------------------------------------------------------------------------------------------</td>
<td>-----------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Automatic Data Typing View</td>
<td>Name, Run, CompiledDT, CompiledDesignMax, CompiledDesignMin, Accept, ProposedDT, SpecifiedDT, DesignMin, DesignMax, DerivedMin, DerivedMax, SimMin, SimMax, OverflowWraps, Saturations, ProposedMin, ProposedMax</td>
<td>After proposing data types if proposal is based on simulation, derived, and design min/max.</td>
</tr>
<tr>
<td>Automatic Data Typing With Simulation Min/Max View</td>
<td>Name, Run, CompiledDT, Accept, ProposedDT, SpecifiedDT, SimMin, SimMax, DesignMin, DesignMax, OverflowWraps, Saturations, ProposedMin, ProposedMax</td>
<td>After proposing data types if the proposal is based on simulation and design min/max.</td>
</tr>
<tr>
<td>Automatic Data Typing With Derived Min/Max View</td>
<td>Name, Run, CompiledDesignMax, CompiledDesignMin, Accept, ProposedDT, SpecifiedDT, DerivedMin, DerivedMax, ProposedMin, ProposedMax</td>
<td>After proposing data types if the proposal is based on design min/max and/or derived min/max.</td>
</tr>
<tr>
<td>Data Collection View</td>
<td>Name, Run, CompiledDT, SpecifiedDT, DerivedMin, DerivedMax, SimMin, SimMax, OverflowWraps, Saturations</td>
<td>After simulating or deriving minimum and maximum values if the results have simulation min/max, derived min/max, and design min/max.</td>
</tr>
<tr>
<td>Derived Min/Max View</td>
<td>Name, Run, CompiledDesignMax, CompiledDesignMin, DerivedMin, DerivedMax</td>
<td>After deriving minimum and maximum values.</td>
</tr>
</tbody>
</table>

By selecting **Show Details**, you can:

- Customize the standard column views
- Create your own column views
• Export and import column views saved in MAT-files, which you can share with other users
• Reset views to factory settings

If you upgrade to a new release of Simulink, and the column views available in the Fixed-Point Tool do not match the views described in the documentation, reset your views to factory settings. When you reset all views, the Model Explorer removes all the custom views you have created. Before you reset views to factory settings, export any views that you will want to use in the future.

You can prevent the Fixed-Point Tool from automatically changing the column view of the contents pane by selecting View > Lock Column View in the Fixed-Point Tool menu. For more information on controlling views, see “Customize Model Explorer Views”.

Changing Column Order and Width

You can alter the order and width of columns that appear in the Contents pane as follows:

• To move a column, click and drag the head of a column to a new location among the column headers.
• To make a column wider or narrower, click and drag the right edge of a column header. If you double-click the right edge of a column header, the column width changes to fit its contents.

Sorting by Columns

By default, the Contents pane displays its contents in ascending order of the Name column. You can alter the order in which the Contents pane displays its rows as follows:

• To sort all the rows in ascending order of another column, click the head of that column.
• To change the order from ascending to descending, simply click again on the head of that column.

Dialog Pane

Use the Dialog pane to view and change properties associated with the system under design.
The Dialog pane includes the following components:

<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>System under design</td>
<td>Displays the system under design for conversion. You can change the selected system by clicking Change.</td>
</tr>
<tr>
<td>Component</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Fixed-point preparation</td>
<td>Contains the <strong>Fixed-Point Advisor</strong> button. Use this button to open the Fixed-Point Advisor to guide you through the tasks to prepare your floating-point model for conversion to fixed point. For more information, see “Fixed-Point Advisor” on page 9-12.</td>
</tr>
<tr>
<td>Configure model settings</td>
<td>Contains default configurations that set up run parameters, such as the run name and data type override settings, by clicking a button. For more information, see “Configure model settings” on page 9-13.</td>
</tr>
<tr>
<td>Range collection</td>
<td>Contains controls to collect simulation or derived minimum and maximum data for your model.</td>
</tr>
<tr>
<td>Automatic data typing</td>
<td>Contains controls to propose and, optionally, accept data type proposals.</td>
</tr>
<tr>
<td>Result Details tab</td>
<td>Use this tab to view data type information about the object selected in the <strong>Contents</strong> pane.</td>
</tr>
</tbody>
</table>

**Tips**

From the Fixed-Point Tool **View** menu, you can customize the layout of the **Dialog** pane. Select:

- **Show Fixed-Point Preparation** to show/hide the **Fixed-Point Advisor** button. By default, the Fixed-Point Tool displays this button.
- **Show Dialog View** to show/hide the **Dialog** pane. By default, the Fixed-Point Tool displays this pane.
- **Settings for selected system** to show/hide the **Settings for selected system** pane. By default, the Fixed-Point Tool displays this pane.

**Fixed-Point Advisor**

Open the Fixed-Point Advisor to guide you through the tasks to prepare a floating-point model for conversion to fixed point. Use the Fixed-Point Advisor if your model contains blocks that do not support fixed-point data types.
Configure model settings

Use the configurations to set up model-wide data type override and instrumentation settings prior to simulation. The Fixed-Point Tool provides:

- Frequently-used factory default configurations
- The ability to add and edit custom configurations

Note The factory default configurations apply to the whole model. You cannot use these shortcuts to configure subsystems.

Factory Defaults

<table>
<thead>
<tr>
<th>Factory Default Configuration</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Range collection using double override</td>
<td>Use this configuration to observe ideal numeric behavior of the model and collect ranges for data type proposals.</td>
</tr>
<tr>
<td></td>
<td>This configuration sets:</td>
</tr>
<tr>
<td></td>
<td>• Run name to DoubleOverride</td>
</tr>
<tr>
<td></td>
<td>• Fixed-point instrumentation mode to Minimums, maximums and overflows</td>
</tr>
<tr>
<td></td>
<td>• Data type override to Double</td>
</tr>
<tr>
<td></td>
<td>• Data type override applies to All numeric types</td>
</tr>
<tr>
<td></td>
<td>By default, a button for this configuration appears in the Configure model settings pane.</td>
</tr>
</tbody>
</table>
Factory Default Configuration | Description
--- | ---
Range collection with specified data types | Use this configuration to collect ranges of actual model and to validate current behavior.

This configuration sets:

- **Run name** to NoOverride
- **Fixed-point instrumentation mode** to Minimums, maximums and overflows
- **Data type override** to Use local settings

By default, a button for this shortcut appears in the **Configure model settings** pane.

Remove overrides and disable range collection | Use this configuration to cleanup settings after finishing fixed-point conversion and to restore maximum simulation speed.

This configuration sets:

- **Fixed-point instrumentation mode** to Off
- **Data type override** to Use local settings

By default, a button for this shortcut appears in the **Configure model settings** pane.

**Advanced settings**

Use **Advanced settings** to add new configurations or edit existing user-defined configurations.

**Run name**

Specifies the run name

If you use a default configuration to set up a run, the Fixed-Point Tool uses the run name associated with this configuration. You can override the run name by entering a new name in this field.
**Tips**

- To store data for multiple runs, provide a different run name for each run. Running two simulations with the same run name overwrites the original run unless you select **Merge results from multiple simulations**.
- You can edit the run name in the Contents pane **Run** column.

For more information, see “Run Management” (Fixed-Point Designer).

**Simulate**

Simulates model and stores results.

**Action**

Simulates the model and stores the results with the run name specified in **Run name**. The Fixed-Point Tool displays the run name in the **Run** column of the **Contents** pane.

**Merge instrumentation results from multiple simulations**

Control how simulation results are stored

**Settings**

**Default**: Off

- **On**
  Merges new simulation minimum and maximum results with existing simulation results in the run specified by the run name parameter. Allows you to collect complete range information from multiple test benches. Does not merge signal logging results.

- **Off**
  Clears all existing simulation results from the run specified by the run name parameter before displaying new simulation results.

**Command-Line Alternative**

**Parameter**: 'MinMaxOverflowArchiveMode'

**Type**: string

**Value**: 'Overwrite' | 'Merge'
**Default:** 'Overwrite'

**Tip**

Select this parameter to log simulation minimum and maximum values captured over multiple simulations. For more information, see “Propose Data Types For Merged Simulation Ranges” (Fixed-Point Designer).

**Derive ranges for selected system**

Derive minimum and maximum values for signals for the selected system.

The Fixed-Point Tool analyzes the selected system to compute derived minimum and maximum values based on design minimum and maximum values specified on blocks. For example, using the **Output minimum** and **Output maximum** for block outputs.

**Action**

Analyzes the selected system to compute derived minimum and maximum information based on the design minimum and maximum values specified on blocks.

By default, the Fixed-Point Tool displays the **Derived Min/Max View** with the following information in the **Contents** pane.

**Command-Line Alternative**

No command line alternative available.

**Dependencies**

Range analysis:

- Requires a Fixed-Point Designer license.

**Propose**

**Signedness**

Select whether you want The Fixed-Point Tool to propose signedness for results in your model. The Fixed-Point Tool proposes signedness based on collected range data and block constraints. By default, the **Signedness** check box is selected.
When the check box is selected, signals that are always strictly positive get an unsigned data type proposal. If you clear the check box, the Fixed-Point Tool proposes a signed data type for all results that currently specify a floating-point or an inherited output data type unless other constraints are present. If a result specifies a fixed-point output data type, the Fixed-Point Tool will propose a data type with the same signedness as the currently specified data type unless other constraints are present.

**Word length or fraction length**

Select whether you want the Fixed-Point Tool to propose word lengths or fraction lengths for the objects in your system.

- If you select **Word length**, the Fixed-Point Tool proposes a data type with the specified fraction length and the minimum word length to avoid overflows.
- If you select **Fraction length**, the Fixed-Point Tool proposes a data type with the specified word length and best-precision fraction length while avoiding overflows.

If a result currently specifies a fixed-point data type, that information will be used in the proposal. If a result specifies a floating-point or inherited output data type, and the **Inherited** and **Floating point** check boxes are selected, the Fixed-Point Tool uses the settings specified under **Automatic data typing** to make a data type proposal.

**Propose for**

**Inherited**

Propose data types for results that specify one of the inherited output data types.

**Floating-point**

Propose data types for results that specify floating-point output data types.

**Default fraction length**

Specify the default fraction length for objects in your model. The Fixed-Point Tool proposes a data type with the specified fraction length and the minimum word length that avoids overflows.

**Command-Line Alternative**

No command line alternative available.
**Default word length**

Specify the default word length for objects in your model. The Fixed-Point Tool will propose best-precision fraction lengths based on the specified default word length.

**Command-Line Alternative**

No command line alternative available.

**When proposing types use**

Specify the types of ranges to use for data type proposals.

**Design and derived ranges**

The Fixed-Point Tool uses the design ranges in conjunction with derived ranges to propose data types. Design ranges take precedence over derived ranges.

**Design and simulation ranges**

The Fixed-Point Tool uses the design ranges in conjunction with collected simulation ranges to propose data types. Design ranges take precedence over simulation ranges.

The **Safety margin for simulation min/max (%)** parameter specifies a range that differs from that defined by the simulation range. For more information, see “Safety margin for simulation min/max (%)” on page 9-18

**All collected ranges**

The Fixed-Point Tool uses design ranges in addition to derived and simulation ranges to propose data types.

Design minimum and maximum values take precedence over simulation and derived ranges.

**Command-Line Alternative**

No command line alternative available.

**Safety margin for simulation min/max (%)**

Specify safety factor for simulation minimum and maximum values.
**Settings**

**Default:** 0

The simulation minimum and maximum values are adjusted by the percentage designated by this parameter, allowing you to specify a range different from that obtained from the simulation run. The specified safety margin must be a real number greater than -100. For example, a value of 55 specifies that a range at least 55 percent larger is desired. A value of -15 specifies that a range up to 15 percent smaller is acceptable.

**Dependencies**

Before performing automatic data typing, you must specify design minimum and maximum values or run a simulation to collect simulation minimum and maximum data, or collect derived minimum and maximum values.

**Command-Line Alternative**

No command line alternative available.
Advanced Settings

In this section...

“Advanced Settings Overview” on page 9-20
“Fixed-point instrumentation mode” on page 9-20
“Data type override” on page 9-21
“Data type override applies to” on page 9-23
“Name of shortcut” on page 9-25
“Allow modification of fixed-point instrumentation settings” on page 9-25
“Allow modification of data type override settings” on page 9-26
“Allow modification of run name” on page 9-27
“Run name” on page 9-27
“Capture system settings” on page 9-27
“Fixed-point instrumentation mode” on page 9-27
“Data type override” on page 9-28
“Data type override applies to” on page 9-29

Advanced Settings Overview

Use the Advanced Settings dialog to control the fixed-point instrumentation mode, and data type override settings. You can also use the Advanced Settings dialog to add or edit user-defined configurations. You cannot modify the factory default configurations. If you add a new configuration and want it to appear as a button on the Fixed-Point Tool Configure model settings pane, use the controls in the Shortcuts tab.

Fixed-point instrumentation mode

Control which objects log minimum, maximum and overflow data during simulation.

Settings

Default: Use local settings
Use local settings
Logs data according to the value of this parameter set for each subsystem. Otherwise, settings for parent systems always override those of child systems.

Minimums, maximums and overflows
Logs minimum value, maximum value, and overflow data for all blocks in the current system or subsystem during simulation.

Overflows only
Logs only overflow data for all blocks in the current system or subsystem.

Force off
Does not log data for any block in the current system or subsystem. Use this selection to work with models containing fixed-point enabled blocks if you do not have a Fixed-Point Designer license.

Tips
• You cannot change the instrumentation mode for linked subsystems or referenced models.

Dependencies
The value of this parameter for parent systems controls min/max logging for all child subsystems, unless Use local settings is selected.

Command-Line Alternative
Parameter: 'MinMaxOverflowLogging'
Type: string
Value: 'UseLocalSettings' | 'MinMaxAndOverflow' | 'OverflowOnly' | 'ForceOff'
Default: 'UseLocalSettings'

Data type override
Control data type override of objects that allow you to specify data types in their dialog boxes.

Settings
Default: Use local settings
The value of this parameter for parent systems controls data type override for all child subsystems, unless Use local settings is selected.

**Use local settings**

Overrides data types according to the setting of this parameter for each subsystem.

**Scaled double**

Overrides the data type of all blocks in the current system and subsystem with doubles; however, the scaling and bias specified in the dialog box of each block is maintained.

**Double**

Overrides the output data type of all blocks in the current system or subsystem with doubles. The overridden values have no scaling or bias.

**Single**

Overrides the output data type of all blocks in the current system or subsystem with singles. The overridden values have no scaling or bias.

**Off**

No data type override is performed on any block in the current system or subsystem. The settings on the blocks are used.

**Tips**

- Set this parameter to Double or Single and the **Data type override applies to** parameter to All numeric types to work with models containing fixed-point enabled blocks if you do not have a Fixed-Point Designer license.
- You cannot change the **Data type override** setting on linked subsystems or referenced models.
- Data type override never applies to boolean data types.
- When you set the **Data type override** parameter of a parent system to Double, Single, Scaled double or Off, this setting also applies to all child subsystems and you cannot change the data type override setting for these child subsystems. When the **Data type override** parameter of a parent system is Use local settings, you can set the **Data type override** parameter for individual children.
- Use this parameter with the **Data type override applies to** parameter. The following table details how these two parameters affect the data types in your model.
### Fixed-Point Tool Settings

<table>
<thead>
<tr>
<th>Data type override</th>
<th>Data type override applies to</th>
<th>Floating-point types</th>
<th>Fixed-point types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use local settings/Off</td>
<td>N/A</td>
<td>Unchanged</td>
<td>Unchanged</td>
</tr>
<tr>
<td>Double</td>
<td>All numeric types</td>
<td>Double</td>
<td>Double</td>
</tr>
<tr>
<td></td>
<td>Floating-point</td>
<td>Double</td>
<td>Unchanged</td>
</tr>
<tr>
<td></td>
<td>Fixed-point</td>
<td>Unchanged</td>
<td>Double</td>
</tr>
<tr>
<td>Single</td>
<td>All numeric types</td>
<td>Single</td>
<td>Single</td>
</tr>
<tr>
<td></td>
<td>Floating-point</td>
<td>Single</td>
<td>Unchanged</td>
</tr>
<tr>
<td></td>
<td>Fixed-point</td>
<td>Unchanged</td>
<td>Single</td>
</tr>
<tr>
<td>Scaled double</td>
<td>All numeric types</td>
<td>Double</td>
<td>Scaled double equivalent of fixed-point type</td>
</tr>
<tr>
<td></td>
<td>Floating-point</td>
<td>Double</td>
<td>Unchanged</td>
</tr>
<tr>
<td></td>
<td>Fixed-point</td>
<td>Unchanged</td>
<td>Scaled double equivalent of fixed-point type</td>
</tr>
</tbody>
</table>

### Dependencies

- The following Simulink blocks allow you to set data types in their block masks, but ignore the **Data type override** setting:
  - Probe
  - Trigger
  - Width

### Command-Line Alternative

**Parameter:** 'DataTypeOverride'  
**Type:** string  
**Value:** 'UseLocalSettings' | 'ScaledDouble' | 'Double' | 'Single' | 'Off'  
**Default:** 'UseLocalSettings'

### Data type override applies to

Specifies which data types the Fixed-Point Tool overrides
**Settings**

**Default:** All numeric types

All numeric types

Data type override applies to all numeric types, floating-point and fixed-point. It does not apply to boolean or enumerated data types.

**Floating-point**

Data type override applies only to floating-point data types, that is, double and single.

**Fixed-point**

Data type override applies only to fixed-point data types, for example, uint8, fixdt.

**Tips**

- Use this parameter with the **Data type override** parameter.
- Data type override never applies to boolean or enumerated data types or to buses.
- When you set the **Data type override** parameter of a parent system to Double, Single, Scaled double or Off, this setting also applies to all child subsystems and you cannot change the data type override setting for these child subsystems. When the **Data type override** parameter of a parent system is Use local setting, you can set the **Data type override** parameter for individual children.
- The following table details how these two parameters affect the data types in your model.

<table>
<thead>
<tr>
<th>Fixed-Point Tool Settings</th>
<th>Block Local Settings</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data type override</td>
<td>Data type override applies to</td>
</tr>
<tr>
<td>Use local settings/Off</td>
<td>N/A</td>
</tr>
<tr>
<td>Double</td>
<td>All numeric types</td>
</tr>
<tr>
<td></td>
<td>Floating-point</td>
</tr>
<tr>
<td></td>
<td>Fixed-point</td>
</tr>
<tr>
<td>Single</td>
<td>All numeric types</td>
</tr>
<tr>
<td></td>
<td>Floating-point</td>
</tr>
<tr>
<td></td>
<td>Fixed-point</td>
</tr>
</tbody>
</table>
### Fixed-Point Tool Settings

<table>
<thead>
<tr>
<th>Data type override</th>
<th>Data type override applies to</th>
<th>Floating-point types</th>
<th>Fixed-point types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scaled double</td>
<td>All numeric types</td>
<td>Double</td>
<td>Scaled double equivalent of fixed-point type</td>
</tr>
<tr>
<td>Floating-point</td>
<td>Double</td>
<td>Unchanged</td>
<td></td>
</tr>
<tr>
<td>Fixed-point</td>
<td>Unchanged</td>
<td></td>
<td>Scaled double equivalent of fixed-point type</td>
</tr>
</tbody>
</table>

#### Dependencies

This parameter is enabled only when **Data type override** is set to **Scaled double**, **Double** or **Single**.

#### Command-Line Alternative

**Parameter:** 'DataTypeOverrideAppliesTo'

**Type:** string

**Value:** 'AllNumericTypes' | 'Floating-point' | 'Fixed-point'

**Default:** 'AllNumericTypes'

### Name of shortcut

Enter a unique name for your shortcut. By default, the Fixed-Point Tool uses this name as the **Run name** for this shortcut.

If the shortcut name already exists, the new settings overwrite the existing settings.

#### See Also

- “Run Management” (Fixed-Point Designer)

### Allow modification of fixed-point instrumentation settings

Select whether to change the model fixed-point instrumentation settings when you apply this shortcut to the model.
**Settings**

**Default:** On

- **On**
  When you apply this shortcut to the model, changes the fixed-point instrumentation settings of the model and its subsystems to the setting defined in this shortcut.

- **Off**
  Does not change the fixed-point instrumentation settings when you apply this shortcut to the model.

**Tip**

If you want to control data type override settings without altering the fixed-point instrumentation settings on your model, clear this option.

**See Also**

- “Run Management” (Fixed-Point Designer)

**Allow modification of data type override settings**

Select whether to change the model data type override settings when you apply this shortcut to the model

**Settings**

**Default:** On

- **On**
  When you apply this shortcut to the model, changes the data type override settings of the model and its subsystems to the settings defined in this shortcut.

- **Off**
  Does not change the fixed-point instrumentation settings when you apply this shortcut to the model.
Allow modification of run name

Select whether to change the run name on the model when you apply this shortcut to the model.

Settings

Default: On

On  
Changes the run name to the setting defined in this shortcut when you apply this shortcut to the model.

Off  
Does not change the run name when you apply this shortcut to the model.

Run name

Specify the run name to use when you apply this shortcut.

By default, the run name uses the name of the shortcut. Run names are case sensitive.

Dependency

Allow modification of run name enables this parameter.

Capture system settings

Copy the model and subsystem fixed-point instrumentation mode and data type override settings into the Shortcut editor.

Fixed-point instrumentation mode

Control which objects in the shortcut editor log minimum, maximum and overflow data during simulation.

This information is stored in the shortcut. To use the current model setting, click Capture system settings.
**Settings**

**Default:** Same as model setting

**Use local settings**
Logs data according to the value of this parameter set for each subsystem. Otherwise, settings for parent systems always override those of child systems.

**Minimums, maximums and overflows**
Logs minimum value, maximum value, and overflow data for all blocks in the current system or subsystem during simulation.

**Overflows only**
Logs only overflow data for all blocks in the current system or subsystem.

**Force off**
Does not log data for any block in the current system or subsystem. Use this selection to work with models containing fixed-point enabled blocks if you do not have a Fixed-Point Designer license.

**Dependency**

Allow modification of fixed-point instrumentation settings enables this parameter.

**Data type override**

Control data type override of objects that allow you to specify data types in their dialog boxes.

This information is stored in the shortcut. To use the current model settings, click Capture system settings.

**Settings**

**Default:** Same as model

The value of this parameter for parent systems controls data type override for all child subsystems, unless Use local settings is selected.

**Use local settings**
Overrides data types according to the setting of this parameter for each subsystem.
Scaled double

Overrides the data type of all blocks in the current system and subsystem with doubles; however, the scaling and bias specified in the dialog box of each block is maintained.

Double

Overrides the output data type of all blocks in the current system or subsystem with doubles. The overridden values have no scaling or bias.

Single

Overrides the output data type of all blocks in the current system or subsystem with singles. The overridden values have no scaling or bias.

Off

No data type override is performed on any block in the current system or subsystem. The settings on the blocks are used.

Dependency

Allow modification of data type override settings enables this parameter.

Data type override applies to

Specifies which data types to override when you apply this shortcut.

This information is stored in the shortcut. To use the current model setting, click Capture system settings.

Settings

Default: All numeric types

All numeric types

Data type override applies to all numeric types, floating-point and fixed-point. It does not apply to boolean or enumerated data types.

Floating-point

Data type override applies only to floating-point data types, that is, double and single.

Fixed-point

Data type override applies only to fixed-point data types, for example, uint8, fixdt.
Dependency

Allow modification of data type override settings enables this parameter.
Model Advisor Checks
Simulink Checks

In this section...

“Simulink Check Overview” on page 10-4
“Migrating to Simplified Initialization Mode Overview” on page 10-5
“Identify unconnected lines, input ports, and output ports” on page 10-5
“Check root model Inport block specifications” on page 10-6
“Check optimization settings” on page 10-7
“Check diagnostic settings ignored during accelerated model reference simulation” on page 10-10
“Check for parameter tunability information ignored for referenced models” on page 10-10
“Check for implicit signal resolution” on page 10-11
“Check for optimal bus virtuality” on page 10-12
“Check for Discrete-Time Integrator blocks with initial condition uncertainty” on page 10-13
“Identify disabled library links” on page 10-14
“Check for large number of function arguments from virtual bus across model reference boundary” on page 10-15
“Identify parameterized library links” on page 10-15
“Identify unresolved library links” on page 10-16
“Identify configurable subsystem blocks for converting to variant subsystem blocks” on page 10-17
“Identify Variant Model blocks and convert those to Variant Subsystem containing Model block choices” on page 10-18
“Identify Variant blocks using Variant objects with empty conditions” on page 10-18
“Check usage of function-call connections” on page 10-19
“Check Data Store Memory blocks for multitasking, strong typing, and shadowing issues” on page 10-19
“Check if read/write diagnostics are enabled for data store blocks” on page 10-21
“Check data store block sample times for modeling errors” on page 10-22
“Check for potential ordering issues involving data store access” on page 10-23
<table>
<thead>
<tr>
<th>In this section...</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Check structure parameter usage with bus signals” on page 10-24</td>
</tr>
<tr>
<td>“Check Delay, Unit Delay and Zero-Order Hold blocks for rate transition” on page 10-26</td>
</tr>
<tr>
<td>“Check for calls to slDataTypeAndScale” on page 10-28</td>
</tr>
<tr>
<td>“Check bus signals treated as vectors” on page 10-29</td>
</tr>
<tr>
<td>“Check for potentially delayed function-call subsystem return values” on page 10-30</td>
</tr>
<tr>
<td>“Identify block output signals with continuous sample time and non-floating point data type” on page 10-31</td>
</tr>
<tr>
<td>“Check usage of Merge blocks” on page 10-32</td>
</tr>
<tr>
<td>“Check usage of Outport blocks” on page 10-35</td>
</tr>
<tr>
<td>“Check usage of Discrete-Time Integrator blocks” on page 10-46</td>
</tr>
<tr>
<td>“Check model settings for migration to simplified initialization mode” on page 10-47</td>
</tr>
<tr>
<td>“Check S-functions in the model” on page 10-49</td>
</tr>
<tr>
<td>“Check for non-continuous signals driving derivative ports” on page 10-50</td>
</tr>
<tr>
<td>“Runtime diagnostics for S-functions” on page 10-51</td>
</tr>
<tr>
<td>“Check model for foreign characters” on page 10-52</td>
</tr>
<tr>
<td>“Identify unit mismatches in the model” on page 10-53</td>
</tr>
<tr>
<td>“Identify automatic unit conversions in the model” on page 10-53</td>
</tr>
<tr>
<td>“Identify disallowed unit systems in the model” on page 10-54</td>
</tr>
<tr>
<td>“Identify undefined units in the model” on page 10-54</td>
</tr>
<tr>
<td>“Check model for block upgrade issues” on page 10-55</td>
</tr>
<tr>
<td>“Check model for block upgrade issues requiring compile time information” on page 10-56</td>
</tr>
<tr>
<td>“Check that the model is saved in SLX format” on page 10-57</td>
</tr>
<tr>
<td>“Check model for SB2SL blocks” on page 10-58</td>
</tr>
<tr>
<td>“Check Model History properties” on page 10-58</td>
</tr>
<tr>
<td>“Identify Model Info blocks that can interact with external source control tools” on page 10-59</td>
</tr>
<tr>
<td>“Identify Model Info blocks that use the Configuration Manager” on page 10-60</td>
</tr>
<tr>
<td>“Check model for legacy 3DoF or 6DoF blocks” on page 10-61</td>
</tr>
</tbody>
</table>
In this section...

- “Check model and local libraries for legacy Aerospace Blockset blocks” on page 10-62
- “Check model for Aerospace Blockset navigation blocks” on page 10-62
- “Check and update masked blocks in library to use promoted parameters” on page 10-63
- “Check and update mask image display commands with unnecessary imread() function calls” on page 10-64
- “Check and update mask to affirm icon drawing commands dependency on mask workspace” on page 10-65
- “Identify masked blocks that specify tabs in mask dialog using MaskTabNames parameter” on page 10-66
- “Identify questionable operations for strict single-precision design” on page 10-67
- “Check get_param calls for block CompiledSampleTime” on page 10-68
- “Check if all simulation outputs are returned as a single Simulink.SimulationOutput object” on page 10-70
- “Check model for parameter initialization and tuning issues” on page 10-71
- “Check for virtual bus across model reference boundaries” on page 10-72
- “Check model for custom library blocks that rely on frame status of the signal” on page 10-74
- “Check model for S-function upgrade issues” on page 10-75
- “Check Rapid accelerator signal logging” on page 10-76
- “Check virtual bus inputs to blocks” on page 10-77
- “Check for root outports with constant sample time” on page 10-81
- “Analyze model hierarchy and continue upgrade sequence” on page 10-81
- “Check Access to Data Stores” on page 10-83
- “Check relative execution orders for Data Store Read and Data Store Write blocks” on page 10-84

Simulink Check Overview

Use the Simulink Model Advisor checks to configure your model for simulation.
See Also

- “Run Model Advisor Checks”
- “Simulink Coder Checks” (Simulink Coder)
- “Simulink Check Checks” (Simulink Check)

Migrating to Simplified Initialization Mode Overview

Simplified initialization mode was introduced in R2008b to improve the consistency of simulation results. This mode is especially important for models that do not specify initial conditions for conditionally executed subsystem output ports. For more information, see “Simplified Initialization Mode” and “Classic Initialization Mode”.

Use the Model Advisor checks in Migrating to Simplified Initialization Mode to help migrate your model to simplified initialization mode.

See Also

- “Simplified Initialization Mode”
- “Classic Initialization Mode”
- “Underspecified initialization detection”
- “Check usage of Merge blocks” on page 10-32
- “Check usage of Outport blocks” on page 10-35
- “Check usage of Discrete-Time Integrator blocks” on page 10-46
- “Check model settings for migration to simplified initialization mode” on page 10-47

Identify unconnected lines, input ports, and output ports

**Check ID:** mathworks.design.UnconnectedLinesPorts

Check for unconnected lines or ports.

**Description**

This check lists unconnected lines or ports. These can have difficulty propagating signal attributes such as data type, sample time, and dimensions.

**Note** Ports connected to ground/terminator blocks will pass this test.
**Results and Recommended Actions**

<table>
<thead>
<tr>
<th>Condition</th>
<th>Recommended Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lines, input ports, or output ports are unconnected.</td>
<td>Connect the signals. Double-click the list of unconnected items to locate failure.</td>
</tr>
</tbody>
</table>

**Capabilities and Limitations**

You can:

- Run this check on your library models.
- Exclude blocks and charts from this check if you have a Simulink Check license.

**Tips**

Use the `PortConnectivity` command to obtain an array of structures describing block input or output ports.

**See Also**

“Common Block Properties” on page 6-112 for information on the `PortConnectivity` command.

“Model Advisor Exclusion Overview” (Simulink Check)

**Check root model Inport block specifications**

**Check ID:** `mathworks.design.RootInportSpec`

Check that root model Inport blocks fully define dimensions, sample time, and data type.

**Description**

Using root model Inport blocks that do not fully define dimensions, sample time, or data type can lead to undesired simulation results. Simulink software back-propagates dimensions, sample times and data types from downstream blocks unless you explicitly assign them values.
Results and Recommended Actions

<table>
<thead>
<tr>
<th>Condition</th>
<th>Recommended Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Root-level Inport blocks have undefined attributes.</td>
<td>Fully define the attributes of the root-level Inport blocks.</td>
</tr>
</tbody>
</table>

Capabilities and Limitations

If you have a Simulink Check license, you can exclude blocks and charts from this check.

Tips

The following configurations pass this check:

• Configuration Parameters > Solver > Periodic sample time constraint is set to Ensure sample time independent
• For export-function models, inherited sample time is not flagged.

See Also

• “About Data Types in Simulink”.
• “Determine Output Signal Dimensions”.
• “Specify Sample Time”.
• “Model Advisor Exclusion Overview” (Simulink Check)

Check optimization settings

Check ID: mathworks.design.OptimizationSettings

Check for optimizations that can lead to non-optimal code generation and simulation.

Description

This check reviews the status of optimizations that can improve code efficiency and simulation time.
## Results and Recommended Actions

<table>
<thead>
<tr>
<th>Condition</th>
<th>Recommended Action</th>
</tr>
</thead>
</table>
| The specified optimizations are off. | Select the following optimization check boxes on the **Optimization** pane in the Configuration Parameters dialog box:  
  • “Remove root level I/O zero initialization” (Simulink Coder)  
  • “Remove internal data zero initialization” (Simulink Coder)  
Select the following optimization check boxes on the **Optimization** pane in the Configuration Parameters dialog box:  
  • “Inline invariant signals” (Simulink Coder) (only if you have a Simulink Coder license)  
Select the following optimization check boxes in the Configuration Parameters dialog box:  
  • “Block reduction”  
  • “Conditional input branch execution”  
  • “Implement logic signals as Boolean data (vs. double)”  
  • “Use memset to initialize floats and doubles to 0.0” (Simulink Coder)  
  • “Remove code from floating-point to integer conversions that wraps out-of-range values” (Simulink Coder) (only if you have a Simulink Coder license)  
  • “Signal storage reuse” (Simulink Coder) (only if you have a Simulink Coder license)  
  • “Enable local block outputs” (Simulink Coder)  
  • “Reuse local block outputs” (Simulink Coder)  
  • “Eliminate superfluous local variables (Expression folding)” (Simulink Coder)  
Select the following optimization check boxes on the **Optimization** pane in the Configuration Parameters dialog box:  

**Note** Model Advisor checks these parameters only if there is a Stateflow chart in the model.
<table>
<thead>
<tr>
<th>Condition</th>
<th>Recommended Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Application lifespan (days)” is set as infinite. This could lead to expensive 64-bit counter usage.</td>
<td>Choose a stop time if this is not intended.</td>
</tr>
</tbody>
</table>
| The specified diagnostics, which can increase the time it takes to simulate your model, are set to warning or error. | Select none for:  
  - **Solver data inconsistency**  
  - **Array bounds exceeded**  
  - **Diagnostics > Data Validity > Simulation range checking**                                                                                     |
| The specified Embedded Coder parameters are off.                         | If you have an Embedded Coder license and you are using an ERT-based system target file:  
  - Select **Single output/update function**. For details, see “Single output/update function” (Simulink Coder).  
  - Select **Ignore test point signals**. For details, see “Ignore test point signals” (Simulink Coder).  
  - Set **Pass reusable subsystem outputs as** to **Individual arguments**. For details, see “Pass reusable subsystem outputs as” (Simulink Coder). |

**Tips**

If the system contains Model blocks and the referenced model is in Accelerator mode, simulating the model requires generating and compiling code.

check

**See Also**

- “Model Configuration Parameters: Code Generation Optimization” (Simulink Coder)
Check diagnostic settings ignored during accelerated model reference simulation

**Check ID:** mathworks.design.ModelRefSIMConfigCompliance

Checks for referenced models for which Simulink changes configuration parameter settings during accelerated simulation.

**Description**

For models referenced in accelerator mode, Simulink ignores the settings of the following configuration parameters that you set to a value other than None.

- **Array bounds exceeded**
- **Diagnostics > Data Validity > Inf or NaN block output**
- **Diagnostics > Data Validity > Division by singular matrix**
- **Diagnostics > Data Validity > Wrap on overflow**

Also, for models referenced in accelerator mode, Simulink ignores the following Configuration Parameters > Diagnostics > Data Validity > Data Store Memory block parameters if you set them to a value other than Disable all. For details, see “Data Store Diagnostics”.

- **Detect read before write**
- **Detect write after read**
- **Detect write after write**

**Results and Recommended Actions**

<table>
<thead>
<tr>
<th>Condition</th>
<th>Recommended Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>You want to see the results of running the identified diagnostics with settings to produce warnings or errors.</td>
<td>Simulate the model in Normal mode and resolve diagnostic warnings or errors.</td>
</tr>
</tbody>
</table>

Check for parameter tunability information ignored for referenced models

**Check ID:** mathworks.design.ParamTunabilityIgnored
Checks if parameter tunability information is included in the Model Parameter Configuration dialog box.

**Description**

Simulink software ignores tunability information specified in the Model Parameter Configuration dialog box. This check identifies those models containing parameter tunability information that Simulink software will ignore if the model is referenced by other models.

**Results and Recommended Actions**

<table>
<thead>
<tr>
<th>Condition</th>
<th>Recommended Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model contains ignored parameter tunability information.</td>
<td>Click the links to convert to equivalent Simulink parameter objects in the MATLAB workspace.</td>
</tr>
</tbody>
</table>

**See Also**

“Create Tunable Calibration Parameter in the Generated Code” (Simulink Coder).

**Check for implicit signal resolution**

**Check ID:** mathworks.design.ImplicitSignalResolution

Identify models that attempt to resolve named signals and states to Simulink.Signal objects.

**Description**

Requiring Simulink software to resolve all named signals and states is inefficient and slows incremental code generation and model reference. This check identifies those signals and states for which you may turn off implicit signal resolution and enforce resolution.
Results and Recommended Actions

<table>
<thead>
<tr>
<th>Condition</th>
<th>Recommended Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Not all signals and states are resolved.</td>
<td>Turn off implicit signal resolution and enforce resolution for each signal and state that does resolve.</td>
</tr>
</tbody>
</table>

See Also

“Resolve Signal Objects for Output Data”.

Check for optimal bus virtuality

Check ID: mathworks.design.OptBusVirtuality

Identify virtual buses that could be made nonvirtual. Making these buses nonvirtual improves generated code efficiency.

Description

This check identifies blocks incorporating virtual buses that cross a subsystem boundary. Changing these to nonvirtual improves generated code efficiency.

Results and Recommended Actions

<table>
<thead>
<tr>
<th>Condition</th>
<th>Recommended Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blocks that specify a virtual bus crossing a subsystem boundary.</td>
<td>Change the highlighted bus to nonvirtual.</td>
</tr>
</tbody>
</table>

Capabilities and Limitations

You can:

- Run this check on your library models.
- Exclude blocks and charts from this check if you have a Simulink Check license.

See Also

- “Signal Basics”
- “Types of Composite Signals”
Check for Discrete-Time Integrator blocks with initial condition uncertainty

Check ID: mathworks.design.DiscreteTimeIntegratorInitCondition

Identify Discrete-Time Integrator blocks with state ports and initial condition ports that are fed by neither an Initial Condition nor a Constant block.

Description

Discrete-Time Integrator blocks with state port and initial condition ports might not be suitably initialized unless they are fed from an Initial Condition or Constant block. This is more likely to happen when Discrete-Time Integrator blocks are used to model second-order or higher-order dynamic systems.

Results and Recommended Actions

<table>
<thead>
<tr>
<th>Condition</th>
<th>Recommended Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Discrete-Time Integrator blocks are not initialized during the model initialization phase.</td>
<td>Add a Constant or Initial Condition block to feed the external Initial Condition port.</td>
</tr>
</tbody>
</table>

Capabilities and Limitations

You can:

- Run this check on your library models.
- Exclude blocks and charts from this check if you have a Simulink Check license.

See Also

- IC block
- Discrete-Time Integrator block
- Constant block
- “Model Advisor Exclusion Overview” (Simulink Check)
Identify disabled library links

Check ID: mathworks.design.DisabledLibLinks

Search model for disabled library links.

Description

Disabled library links can cause unexpected simulation results. Resolve disabled links before saving a model.

Note This check may overlap with “Check model for block upgrade issues” on page 10-55.

Results and Recommended Actions

<table>
<thead>
<tr>
<th>Condition</th>
<th>Recommended Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Library links are disabled.</td>
<td>Click the Library Link &gt; Resolve link option in the context menu.</td>
</tr>
</tbody>
</table>

Capabilities and Limitations

You can:

• Run this check on your library models.
• Exclude blocks and charts from this check if you have a Simulink Check license.

Tips

• Use the Model Browser to find library links.
• To enable a broken link, right-click a block in your model to display the context menu. Select Library Link > Resolve link.

See Also

“Restore Disabled or Parameterized Links”

“Model Advisor Exclusion Overview” (Simulink Check)
Check for large number of function arguments from virtual bus across model reference boundary

**Check ID:** mathworks.design.CheckVirtualBusAcrossModelReferenceArgs

Checks virtual bus signals that cross model reference boundaries and flags cases where using virtual buses across a model reference boundary increases the number of function arguments significantly.

**Description**

To improve the speed of the code generation process, you can use this check to reduce the number of generated function arguments. If the check finds a model that where many arguments will be generated for a function, you can click **Update Model** to modify the model so that it generates fewer arguments.

**Results and Recommended Action**

Methods that generate many function arguments as the result of a virtual bus signal crossing model reference boundary slow down the code generation process.

<table>
<thead>
<tr>
<th>Condition</th>
<th>Recommended Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Methods are listed that generate a large number of arguments for the current the model configuration that this check can reduce by modifying the model.</td>
<td>Click <strong>Update Model</strong>.</td>
</tr>
</tbody>
</table>

Clicking **Update Model** resets Inport and Outport block parameters and inserts Signal Conversion blocks, as necessary, to reduce the number of generated function arguments for the model.

**See Also**

“Bus Data Crossing Model Reference Boundaries”

**Identify parameterized library links**

**Check ID:** mathworks.design.ParameterizedLibLinks

Search model for parameterized library links.
Description

Parameterized library links that are unintentional can result in unexpected parameter settings in your model. This can result in improper model operation.

Results and Recommended Actions

<table>
<thead>
<tr>
<th>Condition</th>
<th>Recommended Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameterized links are listed.</td>
<td>Verify that the links are intended to be parameterized.</td>
</tr>
</tbody>
</table>

Capabilities and Limitations

You can:

- Run this check on your library models.
- Exclude blocks and charts from this check if you have a Simulink Check license.

Tips

- Right-click a block in your model to display the context menu. Choose Link Options and click Go To Library Block to see the original block from the library.
- To parameterize a library link, choose Look Under Mask, from the context menu and select the parameter.

See Also

“Restore Disabled or Parameterized Links”

“Model Advisor Exclusion Overview” (Simulink Check)

Identify unresolved library links

Check ID: mathworks.design.UnresolvedLibLinks

Search the model for unresolved library links, where the specified library block cannot be found.
Description

Check for unresolved library links. Models do not simulate while there are unresolved library links.

Results and Recommended Actions

<table>
<thead>
<tr>
<th>Condition</th>
<th>Recommended Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Library links are unresolved.</td>
<td>Locate missing library block or an alternative.</td>
</tr>
</tbody>
</table>

Capabilities and Limitations

You can:

- Run this check on your library models.
- Exclude blocks and charts from this check if you have a Simulink Check license.

See Also

“Fix Unresolved Library Links”

“Model Advisor Exclusion Overview” (Simulink Check)

Identify configurable subsystem blocks for converting to variant subsystem blocks

Check ID: `mathworks.design.CSStoVSSConvert`

Search the model to identify configurable subsystem blocks at the model or subsystem level.

Results and Recommended Actions

<table>
<thead>
<tr>
<th>Condition</th>
<th>Recommended Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Configurable subsystem blocks are identified.</td>
<td>Convert these blocks to variant subsystem blocks to avoid compatibility issues. See Configurable Subsystem.</td>
</tr>
</tbody>
</table>
Capabilities and Limitations

You can run this check on your library models.

See Also

“Convert to Variant Subsystem” on page 1-228

Identify Variant Model blocks and convert those to Variant Subsystem containing Model block choices

Check ID: mathworks.design.ConvertMdlrefVarToVSS

Search the model to identify Variant Model blocks.

Results and Recommended Actions

<table>
<thead>
<tr>
<th>Condition</th>
<th>Recommended Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variant Model blocks available in the model are listed.</td>
<td>Convert these blocks to Variant Subsystem blocks.</td>
</tr>
</tbody>
</table>

See Also

“Convert to Variants”

Identify Variant blocks using Variant objects with empty conditions

Check ID: mathworks.design.emptyVariantObjects

Search the model to identify the Variant blocks or library having empty Variant objects.

Results and Recommended Actions

<table>
<thead>
<tr>
<th>Condition</th>
<th>Recommended Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>The Variant blocks in the model or library having empty Variant objects are listed.</td>
<td>Use valid conditions in the variant objects.</td>
</tr>
</tbody>
</table>
See Also

“Introduction to Variant Controls”

Check usage of function-call connections

Check ID: mathworks.design.CheckForProperFcnCallUsage

Check model diagnostic settings that apply to function-call connectivity and that might impact model execution.

Description

Check for connectivity diagnostic settings that might lead to non-deterministic model execution.

Results and Recommended Actions

<table>
<thead>
<tr>
<th>Condition</th>
<th>Recommended Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Diagnostics &gt; Connectivity &gt; Invalid function-call connection is set to warning. This might lead to non-deterministic model execution.</td>
<td>Set Diagnostics &gt; Connectivity &gt; Invalid function-call connection to error.</td>
</tr>
<tr>
<td>Diagnostic &gt; Connectivity &gt; Context-dependent inputs is set to Disable All or Use local settings. This might lead to non-deterministic model execution.</td>
<td>Set Diagnostics &gt; Connectivity &gt; Context-dependent inputs to Enable all as errors.</td>
</tr>
</tbody>
</table>

See Also

Function-Call Subsystem

Check Data Store Memory blocks for multitasking, strong typing, and shadowing issues

Check ID: mathworks.design.DataStoreMemoryBlkIssue

Look for modeling issues related to Data Store Memory blocks.
Description
Checks for multitasking data integrity, strong typing, and shadowing of data stores of higher scope.

Results and Recommended Actions

<table>
<thead>
<tr>
<th>Condition</th>
<th>Recommended Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>The <strong>Duplicate data store names</strong> check is set to none or warning.</td>
<td>Consider setting the “Duplicate data store names” check to <strong>error</strong> in the Configuration Parameters dialog box, on the <strong>Diagnostics &gt; Data Validity</strong> pane.</td>
</tr>
<tr>
<td>The data store variable names are not strongly typed in one of the following: - <strong>Signal Attributes</strong> pane of the <strong>Block Parameters</strong> dialog for the Date Store Memory block - Global data store name</td>
<td>Specify a data type other than auto by taking one of the following actions: - Choose a data type other than <strong>Inherit: auto</strong> on the <strong>Signal Attributes</strong> pane of the <strong>Block Parameters</strong> dialog for the Date Store Memory block. - If you are using a global data store name, then specify its data type in the Simulink.Signal object.</td>
</tr>
<tr>
<td>The <strong>Multitask data store</strong> check is set to none or warning.</td>
<td>Consider setting the “Multitask data store” check to <strong>error</strong> in the Configuration Parameters dialog box, on the <strong>Diagnostics &gt; Data Validity</strong> pane.</td>
</tr>
</tbody>
</table>

Capabilities and Limitations
If you have a Simulink Check license, you can exclude blocks and charts from this check.

See Also
- “Local and Global Data Stores”
- “Storage Classes for Data Store Memory Blocks” (Simulink Coder)
- Data Store Memory
- Data Store Read
- Data Store Write
- “Duplicate data store names”
Check if read/write diagnostics are enabled for data store blocks

Check ID: mathworks.design.DiagnosticDataStoreBlk

For data store blocks in the model, enable the read-and-write diagnostics order checking to detect run-time issues.

Description

Check for the read-and-write diagnostics order checking. By enabling the read-and-write diagnostics, you detect potential run-time issues.

Results and Recommended Actions

<table>
<thead>
<tr>
<th>Condition</th>
<th>Recommended Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>The <strong>Detect read before write</strong> check is disabled.</td>
<td>Consider enabling “Detect read before write” in the Configuration Parameter dialog box <strong>Diagnostics</strong>&gt; <strong>Data Validity</strong> pane.</td>
</tr>
<tr>
<td>The <strong>Detect write after read</strong> check is disabled.</td>
<td>Consider enabling “Detect write after read” in the Configuration Parameter dialog box <strong>Diagnostics</strong>&gt; <strong>Data Validity</strong> pane.</td>
</tr>
<tr>
<td>The <strong>Detect write after write</strong> check is disabled.</td>
<td>Consider enabling “Detect write after write” in the Configuration Parameter dialog box <strong>Diagnostics</strong>&gt; <strong>Data Validity</strong> pane.</td>
</tr>
</tbody>
</table>

Capabilities and Limitations

Exclude blocks and charts from this check if you have a Simulink Check license.

Tips

.
The run-time diagnostics can slow simulations down considerably. Once you have verified that Simulink does not generate warnings or errors during simulation, set them to **Disable all**.

**See Also**

- “Local and Global Data Stores”
- Data Store Memory
- Data Store Read
- Data Store Write
- “Detect read before write”
- “Detect write after read”
- “Detect write after write”
- “Check for potential ordering issues involving data store access” on page 10-23
- “Model Advisor Exclusion Overview” (Simulink Check)

**Check data store block sample times for modeling errors**

**Check ID:** mathworks.design.DataStoreBlkSampleTime

Identify modeling errors due to the sample times of data store blocks.

**Description**

Check data store blocks for continuous or fixed-in-minor-step sample times.

**Results and Recommended Actions**

<table>
<thead>
<tr>
<th>Condition</th>
<th>Recommended Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data store blocks in your model have continuous or fixed-in-minor-step sample times.</td>
<td>Consider making the listed blocks discrete or replacing them with either Memory or Goto and From blocks.</td>
</tr>
</tbody>
</table>

**Capabilities and Limitations**

If you have a Simulink Check license, you can exclude blocks and charts from this check.
See Also

- “Local and Global Data Stores”
- Data Store Memory
- Data Store Read
- Data Store Write
- “Fixed-in-Minor-Step”
- “Model Advisor Exclusion Overview” (Simulink Check)

Check for potential ordering issues involving data store access

**Check ID:** mathworks.design.OrderingDataStoreAccess

Look for read/write issues which may cause inaccuracies in the results.

**Description**

During an **Update Diagram**, identify potential issues relating to read-before-write, write-after-read, and write-after-write conditions for data store blocks.

**Results and Recommended Actions**

<table>
<thead>
<tr>
<th>Condition</th>
<th>Recommended Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reading and writing (read-before-write or write-after-read condition) occur out of order.</td>
<td>Consider restructuring your model so that the Data Store Read block executes before the Data Store Write block.</td>
</tr>
<tr>
<td>Multiple writes occur within a single time step.</td>
<td>Change the model to write data only once per time step or refer to the following Tips section.</td>
</tr>
</tbody>
</table>

**Capabilities and Limitations**

If you have a Simulink Check license, you can exclude blocks and charts from this check.

**Tips**

This check performs a static analysis which might not identify every instance of improper usage. Specifically, Function-Call Subsystems, Stateflow Charts, MATLAB for code
generation, For Iterator Subsystems, and For Each Subsystems can cause both missed detections and false positives. For a more comprehensive check, consider enabling the following diagnostics on the Diagnostics > Data Validity pane in the Configuration Parameters dialog box: “Detect read before write”, “Detect write after read”, and “Detect write after write”.

See Also

• “Local and Global Data Stores”
• Data Store Memory
• Data Store Read
• Data Store Write
• “Detect read before write”
• “Detect write after read”
• “Detect write after write”
• “Model Advisor Exclusion Overview” (Simulink Check)

Check structure parameter usage with bus signals

Check ID: mathworks.design.MismatchedBusParams

Identify blocks and Simulink.Signal objects that initialize bus signals by using mismatched structures.

Description

In a model, you can use a MATLAB structure to initialize a bus signal. For example, if you pass a bus signal through a Unit Delay block, you can set the Initial condition parameter to a structure. For basic information about initializing buses by using structures, see “Specify Initial Conditions for Bus Signals”.

Run this check to generate efficient and readable code by matching the shape and numeric data types of initial condition structures with those of bus signals. Matching these characteristics avoids unnecessary explicit typecasts and replaces field-by-field structure assignments with, for example, calls to memcpy.

Partial Structures

This check lists blocks and Simulink.Signal objects that initialize bus signals by using partial structures. During the iterative process of creating a model, you can use partial
structures to focus on a subset of signal elements in a bus. For a mature model, use full structures to:

- Generate readable and efficient code.
- Support a modeling style that explicitly initializes unspecified signals. When you use partial structures, Simulink implicitly initializes unspecified signals.

For more information about full and partial structures, see “Create Full Structures for Initialization” and “Create Partial Structures for Initialization”.

**Data Type Mismatches**

This check lists blocks and Simulink.Signal objects whose initial condition structures introduce data type mismatches. The fields of these structures have numeric data types that do not match the data types of the corresponding bus signal elements.

When you configure an initial condition structure to appear as a tunable global structure in the generated code, avoid unnecessary explicit typecasts by matching the data types. See “Generate Tunable Initial Condition Structure for Bus Signal” (Simulink Coder).

**Results and Recommended Actions**

<table>
<thead>
<tr>
<th>Condition</th>
<th>Recommended Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block or signal object uses partial structure</td>
<td>Consider using the function Simulink.Bus.createMATLABStructure to create a full initial condition structure.</td>
</tr>
<tr>
<td>Data types of structure fields do not match data types of corresponding signal elements</td>
<td>Consider defining the structure as a Simulink.Parameter object, and creating a Simulink.Bus object to use as the data type of the bus signal and of the parameter object. To control numeric data types, use the Simulink.BusElement objects in the bus object.</td>
</tr>
</tbody>
</table>

**See Also**

- “Specify Initial Conditions for Bus Signals”
- “Generate Tunable Initial Condition Structure for Bus Signal” (Simulink Coder)
- “Data Stores with Signal Objects”
- Simulink.Bus.createMATLABStruct
Check Delay, Unit Delay and Zero-Order Hold blocks for rate transition

**Check ID:** mathworks.design.ReplaceZOHDelayByRTB

Identify Delay, Unit Delay, or Zero-Order Hold blocks that are used for rate transition. Replace these blocks with actual Rate Transition blocks.

**Description**

If a model uses Delay, Unit Delay, or Zero-Order Hold blocks to provide rate transition between input and output signals, Simulink makes a hidden replacement of these blocks with built-in Rate Transition blocks. In the compiled block diagram, a yellow symbol and the letters “RT” appear in the upper-left corner of a replacement block. This replacement can affect the behavior of the model, as follows:

- These blocks lose their algorithmic design properties to delay a signal or implement zero-order hold. Instead, they acquire rate transition behavior.

- This modeling technique works only in specific transition configurations (slow-to-fast for Delay and Unit Delay blocks, and fast-to-slow for Zero-Order Hold block). Set the block sample time to be equal to the slower rate (source for the Delay and Unit Delay blocks and destination for the Zero-Order Hold block).

- When the block sample time of a downstream or upstream block changes, these Delay, Unit Delay and Zero-Order Hold blocks might not perform rate transition. For example, setting the source and destination sample times equal stops rate transition. The blocks then assume their original algorithmic design properties.

- The block sample time shows incomplete information about sample time rates. The block code runs at two different rates to handle data transfer. However, the block sample time and sample time color show it as a single-rate block. Tools and MATLAB scripts that use sample time information base their behavior on this information.

An alternative is to replace Delay, Unit Delay, or Zero-Order Hold blocks with actual Rate Transition blocks.

- The technique ensures unambiguous results in block behavior. Delay, Unit Delay, or Zero-Order Hold blocks act according to their algorithmic design to delay and hold signals respectively. Only Rate Transition blocks perform actual rate transition.
• Using an actual Rate Transition block for rate transition offers a configurable solution to handle data transfer if you want to specify deterministic behavior or the type of memory buffers to implement.

Use this check to identify instances in your model where Delay, Unit Delay or Zero-Order Hold blocks undergo hidden replacement to provide rate transition between signals. Click **Upgrade Model** to replace these blocks with actual Rate Transition blocks.

**Results and Recommended Actions**

<table>
<thead>
<tr>
<th>Condition</th>
<th>Recommended Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model has no instances of Delay, Unit Delay, or Zero-Order Hold blocks used for rate transition.</td>
<td>No action required.</td>
</tr>
<tr>
<td>Model has instances of Delay, Unit Delay, or Zero-Order Hold blocks used for rate transition.</td>
<td>The check identifies these instances and allows you to upgrade the model.</td>
</tr>
<tr>
<td></td>
<td>1 Click <strong>Upgrade Model</strong> to replace with actual Rate Transition blocks.</td>
</tr>
<tr>
<td></td>
<td>2 Save changes to your model.</td>
</tr>
</tbody>
</table>

If you do not choose to replace the Delay, Unit Delay, and/or Zero-Order Hold blocks with actual Rate Transition blocks, Simulink continues to perform a hidden replacement of these blocks with built-in rate transition blocks.

**Capabilities and Limitations**

You can:

• Run this check on your library models.
• Exclude blocks and charts from this check if you have a Simulink Check license.

**See Also**

• “Run Model Advisor Checks”
• “Model Upgrades”
• Rate Transition
• “Model Advisor Exclusion Overview” (Simulink Check)
Check for calls to slDataTypeAndScale

**Check ID:** mathworks.design.CallslDataTypeAndScale

Identify calls to the internal function slDataTypeAndScale.

**Description**

In some previous versions of Simulink, opening a model that had been saved in an earlier version triggers an automatic upgrade to code for data type handling. The automatic upgrade inserts calls to the internal function slDataTypeAndScale. Although Simulink continues to support some uses of the function, if you eliminate calls to it, you get cleaner and faster code.

Simulink does not support calls to slDataTypeAndScale when:

- The first argument is a Simulink.AliasType object.
- The first argument is a Simulink.NumericType object with property IsAlias set to true.

Running **Check for calls to slDataTypeAndScale** identifies calls to slDataTypeAndScale that are required or recommended for replacement. In most cases, running the check and following the recommended action removes the calls. You can ignore calls that remain. Run the check unless you are sure there are not calls to slDataTypeAndScale.

**Results and Recommended Actions**

<table>
<thead>
<tr>
<th>Condition</th>
<th>Recommended Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Required Replacement Cases</td>
<td>Manually or automatically replace calls to slDataTypeAndScale. Cases listed require you to replace calls to slDataTypeAndScale.</td>
</tr>
<tr>
<td>Recommended Replacement Cases</td>
<td>For the listed cases, it is recommended that you manually or automatically replace calls to slDataTypeAndScale.</td>
</tr>
<tr>
<td>Manual Inspection Cases</td>
<td>Inspect each listed case to determine whether it should be manually upgraded.</td>
</tr>
</tbody>
</table>

**Capabilities and Limitations**

If you have a Simulink Check license, you can exclude blocks and charts from this check.
Tips

- Do not manually insert a call to `slDataTypeAndScale` into a model. The function was for internal use only.
- Running **Check for calls to slDataTypeAndScale** calls the Simulink function `slRemoveDataTypeAndScale`. Calling this function directly provides a wider range of conversion options. However, you very rarely need more conversion options.

See Also

- For more information about upgrading data types and scales, in the MATLAB Command Window, execute the following:
  - `help slDataTypeAndScale`
  - `help slRemoveDataTypeAndScale`
  - “Model Advisor Exclusion Overview” (Simulink Check)

Check bus signals treated as vectors

**Check ID:** `mathworks.design.BusTreatedAsVector`

Identify bus signals that Simulink treats as vectors.

Description

You cannot use bus signals that the Simulink software implicitly converts to vectors. Instead, either insert a Bus to Vector conversion block between the bus signal and the block input port that it feeds, or use the `Simulink.BlockDiagram.addBusToVector` command.

Results and Recommended Actions

<table>
<thead>
<tr>
<th>Condition</th>
<th>Recommended Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bus signals are implicitly converted to vectors.</td>
<td>Use <code>Simulink.BlockDiagram.addBusToVector</code> or insert a Bus to Vector block.</td>
</tr>
<tr>
<td>Model is not configured to identify bus signals that Simulink treats as vectors.</td>
<td>In the Configuration Parameters dialog box, on the Diagnostics &gt; Connectivity pane, set <strong>Bus signal treated as vector</strong> to error.</td>
</tr>
</tbody>
</table>
**Action Results**

Clicking **Modify** inserts a Bus to Vector block at the input ports of blocks that implicitly convert bus signals to vectors.

**Tips**

- Run this check before running **Check consistency of initialization parameters for Outport and Merge blocks**.
- For more information, see “Correct Buses Used as Vectors”.

**See Also**

- “Correct Buses Used as Vectors”
- Bus to Vector block
- “Bus signal treated as vector”
- “Migrating to Simplified Initialization Mode Overview” on page 10-5
- `Simulink.BlockDiagram.addBusToVector`

**Check for potentially delayed function-call subsystem return values**

**Check ID**: `mathworks.design.DelayedFcnCallSubsys`

Identify function-call return values that might be delayed because Simulink software inserted an implicit Signal Conversion block.

**Description**

So that signals reside in contiguous memory, Simulink software can automatically insert an implicit Signal Conversion block in front of function-call initiator block input ports. This can result in a one-step delay in returning signal values from calling function-call subsystems. The delay can be avoided by ensuring the signal originates from a signal block within the function-call system. Or, if the delay is acceptable, insert a Unit Delay block in front of the affected input ports.
**Results and Recommended Actions**

<table>
<thead>
<tr>
<th>Condition</th>
<th>Recommended Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>The listed block input ports could have an implicit Signal Conversion block.</td>
<td>Decide if a one-step delay in returning signal values is acceptable for the listed signals.</td>
</tr>
<tr>
<td></td>
<td>• If the delay is not acceptable, rework your model so that the input signal originates from within the calling subsystem.</td>
</tr>
<tr>
<td></td>
<td>• If the delay is acceptable, insert a Unit Delay block in front of each listed input port.</td>
</tr>
</tbody>
</table>

**Capabilities and Limitations**

If you have a Simulink Check license, you can exclude blocks and charts from this check.

**See Also**

Signal Conversion block

Unit Delay block

“Model Advisor Exclusion Overview” (Simulink Check)

**Identify block output signals with continuous sample time and non-floating point data type**

**Check ID:** `mathworks.design.OutputSignalSampleTime`

Find continuous sample time, non-floating-point output signals.

**Description**

Non-floating-point signals might not represent continuous variables without loss of information.
Results and Recommended Actions

<table>
<thead>
<tr>
<th>Condition</th>
<th>Recommended Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signals with continuous sample times have a non-floating-point data type.</td>
<td>On the identified signals, either change the sample time to be discrete or fixed-in-minor-step ([0,1]).</td>
</tr>
</tbody>
</table>

Capabilities and Limitations

If you have a Simulink Check license, you can exclude blocks and charts from this check.

See Also

“What Is Sample Time?“.

“Model Advisor Exclusion Overview” (Simulink Check)

Check usage of Merge blocks

Check ID: mathworks.design.MergeBlkUsage

Identify Merge blocks with parameter settings that can lead to unexpected behavior, and help migrate your model to simplified initialization mode.

Note Run this check along with the other checks in the “Migrating to Simplified Initialization Mode Overview” on page 10-5.

Description

Simplified initialization mode was introduced in R2008b to improve the consistency of simulation results. For more information, see “Simplified Initialization Mode” and “Classic Initialization Mode”.

This Model Advisor check identifies settings in the Merge blocks in your model that can cause problems if you use classic initialization mode. It also recommends settings for consistent behavior of Merge blocks. The results of the subchecks contain two types of statements: Failed and Warning. Failed statements identify issues that you must address manually before you can migrate the model to the simplified initialization mode. Warning statements identify issues or changes in behavior that can occur after migration.
## Results and Recommended Actions

<table>
<thead>
<tr>
<th>Condition</th>
<th>Recommended Action</th>
</tr>
</thead>
</table>
| Check the run-time diagnostic setting of the Merge block. | **1** In the Configuration Parameters dialog box, set “Detect multiple driving blocks executing at the same time step” to **error**.  
**2** Verify that the model simulates without errors before running this check again. |
| Check for Model blocks that are using the PIL simulation mode. | The simplified initialization mode does not support the Processor-in-the-loop (PIL) simulation for model references. |
| Check for library blocks with instances that cannot be migrated. | Examine the failed subcheck results for each block to determine the corrective actions. |
| Check for single-input Merge blocks. | Replace both the Mux block used to produce the input signal and the Merge block with one multi-input Merge block.  
Single-input Merge blocks are not supported in the simplified initialization mode. |
| Check for root Merge blocks that have an unspecified **Initial output** value. | If you do not specify an explicit value for the **Initial output** parameter of root Merge blocks, then Simulink uses the default initial value of the output data type.  
A root Merge block is a Merge block with an output port that does not connect to another Merge block. For information on the default initial value, see “Initializing Signal Values”. |
<table>
<thead>
<tr>
<th>Condition</th>
<th>Recommended Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Check for Merge blocks with nonzero input port offsets.</td>
<td>Clear the <strong>Allow unequal port widths</strong> parameter of the Merge block.</td>
</tr>
<tr>
<td>Note</td>
<td>Consider using Merge blocks only for signal elements that require true merging. You can combine other elements with merged elements using the Concatenate block.</td>
</tr>
<tr>
<td>Check for Merge blocks that have unconnected inputs or that have inputs from non-conditionally executed subsystems.</td>
<td>Set the <strong>Number of inputs</strong> parameter of the Merge block to the number of Merge block inputs. You must connect each input to a signal.</td>
</tr>
<tr>
<td></td>
<td>Verify that each Merge block input is driven by a conditionally executed subsystem. Merge blocks cannot be driven directly by an Iterator Subsystem or a block that is not a conditionally executed subsystem.</td>
</tr>
<tr>
<td>Check for Merge blocks with inputs that are combined or reordered outside of conditionally executed subsystems.</td>
<td>Verify that combinations or reordering of Merge block input signals takes place within a conditionally executed subsystem. Such designs may use Mux, Bus Creator, or Selector blocks.</td>
</tr>
<tr>
<td>Check for Merge blocks with inconsistent input sample times.</td>
<td>Verify that input signals to each Merge block have the same <strong>Sample time</strong>.</td>
</tr>
<tr>
<td></td>
<td>Failure to do so could result in unpredictable behavior. Consequently, the simplified initialization mode does not allow inconsistent sample times.</td>
</tr>
<tr>
<td>Check for Merge blocks with multiple input ports that are driven by a single source.</td>
<td>Verify that the Merge block does not have multiple input signals that are driven by the same conditionally executed subsystem or conditionally executed Model block.</td>
</tr>
</tbody>
</table>
## Condition

Check for Merge blocks that use signal objects to specify the **Initial output** value.

## Recommended Action

Verify that the following behavior is acceptable.

In the simplified initialization mode, signal objects cannot specify the **Initial output** parameter of the Merge block. While you can still initialize the output signal for a Merge block using a signal object, the initialization result may be overwritten by that of the Merge block.

**Note** Simulink generates a warning that the initial value of the signal object has been ignored.

### See Also

- “Migrating to Simplified Initialization Mode Overview” on page 10-5
- “Model Advisor Exclusion Overview” (Simulink Check)

## Check usage of Outport blocks

### Check ID: mathworks.design.InitParamOutportMergeBlk

Identify Outport blocks and conditional subsystems with parameter settings that can lead to unexpected behavior, and help migrate your model to simplified initialization mode.

**Note** Run this check along with the other checks in the “Migrating to Simplified Initialization Mode Overview” on page 10-5.

### Description

Simplified initialization mode was introduced in R2008b to improve the consistency of simulation results. This mode is especially important for models that do not specify initial conditions for conditionally executed subsystem output ports. For more information, see “Simplified Initialization Mode” and “Classic Initialization Mode”.

10-35
This Model Advisor check identifies Outport blocks and conditional subsystems in your model that can cause problems if you use the simplified initialization mode. It also recommends settings for consistent behavior of Outport blocks. The results of the subchecks contain two types of statements: Failed and Warning. Failed statements identify issues that you must address manually before you can migrate the model to the simplified initialization mode. Warning statements identify issues or changes in behavior can occur after migration.

**Results and Recommended Actions**

<table>
<thead>
<tr>
<th>Condition</th>
<th>Recommended Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Check for blocks inside of the Iterator Subsystem that require elapsed time.</td>
<td>Within an Iterator Subsystem hierarchy, do not use blocks that require a service that maintains the time that has elapsed between two consecutive executions. Since an Iterator Subsystem can execute multiple times at a given time step, the concept of elapsed time is not well-defined between two such executions. Using these blocks inside of an Iterator Subsystem can cause unexpected behavior.</td>
</tr>
<tr>
<td><strong>Condition</strong></td>
<td><strong>Recommended Action</strong></td>
</tr>
<tr>
<td>---------------</td>
<td>------------------------</td>
</tr>
</tbody>
</table>
| Check for Outport blocks that have conflicting signal buffer requirements. | The Outport block has a function-call trigger or function-call data dependency signal passing through it, along with standard data signals. Some of the standard data signals require an explicit signal buffer for the initialization of the output signal of the corresponding subsystem. However, buffering function-call related signals lead to a function-call data dependency violation.  
Consider modifying the model to pass function-call related signals through a separate Outport block. For examples of function-call data dependency violations, see the example model `sl_subsys_semantics`.  
A standard data signal may require an additional signal copy for one of the following reasons:  
• The Outport block is driven by a block with output that cannot be overwritten. The Ground block and the Constant block are examples of such blocks.  
• The Outport block shares the same signal source with another Outport block in the same subsystem or in one nested within the current subsystem but having a different initial output value.  
• The Outport block connects to the input of a Merge block  
• One of the input signals of the Outport block is specifying a `Simulink.Signal` object with an explicit initial value. |
<table>
<thead>
<tr>
<th>Condition</th>
<th>Recommended Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Check for Outport blocks that are driven by a bus signal and whose <strong>Initial output</strong> value is not scalar.</td>
<td>For Outport blocks driven by bus signals, classic initialization mode does not support <strong>Initial Condition</strong> (IC) structures, while simplified initialization mode does. Hence, when migrating a model from classic to simplified mode, specify a scalar for the <strong>Initial Output</strong> parameter. After migration completes, to specify different initial values for different elements of the bus signal, use IC structures. For more information, see “Create Initial Condition Structures”.</td>
</tr>
</tbody>
</table>
| Check for Outport blocks that require an explicit signal copy.             | An explicit copy of the bus signal driving the Outport block is required for the initialization of the output signal of the corresponding subsystem. Insert a Signal Conversion block before the Outport block, then set the **Output** parameter of the Signal Conversion block to **Bus copy**. A standard data signal may require an additional signal copy for one or more of the following reasons:  
  - A block with output that cannot be overwritten is driving the Outport block. The Ground block and the Constant block are examples of such blocks.  
  - The Outport block shares the same signal source with another Outport block in the same subsystem or in one nested within the current subsystem but having a different initial output value.  
  - The Outport block connects to the input of a Merge block  
  - One of the input signals of the Outport block is specifying a Simulink.Signal object with an explicit initial value. |
<table>
<thead>
<tr>
<th>Condition</th>
<th>Recommended Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Check for merged Outport blocks that inherit the <strong>Initial Output</strong> value from Outport blocks that have been configured to reset when the blocks become disabled.</td>
<td>When Outport blocks are driving a Merge block, do not set their <strong>Output when disabled</strong> parameters to <strong>reset</strong>.</td>
</tr>
<tr>
<td>Check for merged Outport blocks that are driven by nested conditionally executed subsystems.</td>
<td>Determine if the new behavior of the Outport blocks is acceptable. If it is not acceptable, modify the model to account for the new behavior before migrating to the simplified initialization mode.</td>
</tr>
<tr>
<td>Check for merged Outport blocks that reset when the blocks are disabled.</td>
<td>Set the <strong>Output when disabled</strong> parameter of the Outport block to <strong>held</strong>. This setting is required because the Outport block connects to a Merge block. For more information, see Outport.</td>
</tr>
<tr>
<td>Check for Outport blocks that have an undefined <strong>Initial output</strong> value with invalid initial condition sources.</td>
<td>Verify that the following behavior is acceptable. When the <strong>Initial output</strong> parameter is unspecified ([ ]), it inherits the initial output from the source blocks. If at least one of the sources of the Outport block is not a valid source to inherit the initial value, the block uses the default initial value for that data type. For simplified initialization mode, valid sources an Outport blocks can inherit the <strong>Initial output</strong> value from are: Constant, Initial Condition, Merge (with initial output), Stateflow chart, function-call model reference, or conditionally executed subsystem blocks.</td>
</tr>
<tr>
<td>Condition</td>
<td>Recommended Action</td>
</tr>
<tr>
<td>--------------------------------------------------------------------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Check Outport blocks that have automatic rate transitions.</td>
<td>Simulink has inserted a Rate Transition block at the input of the Outport block. Specify the <strong>Initial output</strong> parameter for each Outport block.</td>
</tr>
<tr>
<td></td>
<td>Otherwise, perform the following procedure:</td>
</tr>
<tr>
<td></td>
<td>1 In the Configuration Parameters dialog box, on the <strong>Solver</strong> pane, clear the option “Automatically handle rate transition for data transfer”.</td>
</tr>
<tr>
<td></td>
<td>2 Run this Model Advisor check again.</td>
</tr>
<tr>
<td>Check Outport blocks that have a special signal storage requirement and have an undefined <strong>Initial output</strong> value.</td>
<td>Verify that the following behavior is acceptable.</td>
</tr>
<tr>
<td></td>
<td>Specify the <strong>Initial output</strong> parameter for the Outport block. Set this value to [ ] (empty matrix) to use the default initial value of the output data type.</td>
</tr>
<tr>
<td>Check the <strong>Initial output</strong> setting of Outport blocks that reset when they are disabled.</td>
<td>Specify the <strong>Initial output</strong> parameter of the Outport block.</td>
</tr>
<tr>
<td></td>
<td>You must specify the <strong>Initial output</strong> value for blocks that are configured to reset when they become disabled.</td>
</tr>
<tr>
<td>Check the <strong>Initial output</strong> setting for Outport blocks that pass through a function-call data dependency signal.</td>
<td>You cannot specify an <strong>Initial output</strong> value for the Outport block because function-call data dependency signals are passing through it. To set the <strong>Initial output</strong> value:</td>
</tr>
<tr>
<td></td>
<td>1 Set the <strong>Initial output</strong> parameter of the Outport block to [ ].</td>
</tr>
<tr>
<td></td>
<td>2 Provide the initial value at the source of the data dependency signal rather than at the Outport block.</td>
</tr>
<tr>
<td>Condition</td>
<td>Recommended Action</td>
</tr>
<tr>
<td>---------------------------------------------------------------------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Check for Outport blocks that use signal objects to specify the Initial output value.</td>
<td>Verify that the following behavior is acceptable. In the simplified initialization mode, signal objects cannot specify the Initial output parameter of an Outport block. You can still initialize the input or output signals for an Outport block using signal objects, but the initialization results may be overwritten by those of the Outport block.</td>
</tr>
<tr>
<td>Note If you are working with a conditionally executed subsystem Outport block, Simulink generates a warning that the initial value of the signal object has been ignored.</td>
<td></td>
</tr>
<tr>
<td>Check for library blocks with instances that have warnings.</td>
<td>Examine the warning subcheck results for each block before migrating to the simplified initialization mode.</td>
</tr>
<tr>
<td>Check for merged Outport blocks that are either unconnected or connected to a Ground block.</td>
<td>Verify that the following behavior is acceptable. The Outport block is driving a Merge block, but its inputs are either unconnected or connected to Ground blocks. In the classic initialization mode, unconnected or grounded outports do not update the merge signal even when their parent conditionally executed subsystems are executing. In the simplified initialization mode, however, these outports will update the merge signal with a value of zero when their parent conditionally executed subsystems are executing.</td>
</tr>
<tr>
<td>Condition</td>
<td>Recommended Action</td>
</tr>
<tr>
<td>---------------------------------------------------------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| Check for Outport blocks that obtain the **Initial output** value from an input signal when they are migrated. | Verify that the following behavior is acceptable.                                                                                                                                         
<p>|                                                                           | The <strong>Initial output</strong> parameter of the Outport block is not specified. As a result, the simplified initialization mode will assume that the <strong>Initial output</strong> value for the Outport block is derived from the input signal. This assumption may result in different initialization behavior. |
|                                                                           | If this behavior is not acceptable, modify your model before you migrate to the simplified initialization mode.                                                                                         |
| Check for outer Outport blocks that have an explicit <strong>Initial output</strong>.   | Verify that the following behavior is acceptable.                                                                                                                                               |
|                                                                           | In classic initialization mode, the <strong>Initial output</strong> and <strong>Output when disabled</strong> parameters of the Outport block must match those of their source Outport blocks.                                                 |
|                                                                           | In simplified initialization mode, Simulink sets the <strong>Initial output</strong> parameter of outer Outport blocks to [] (empty matrix) and <strong>Output when disabled</strong> parameter to <strong>held</strong>.                        |
| Check for conditionally executed subsystems that propagate execution context across the output boundary. | Verify that the following behavior is acceptable.                                                                                                                                               |
|                                                                           | The <strong>Propagate execution context across subsystem boundary</strong> parameter is selected for the subsystem. Execution context will still be propagated across input boundaries; however, the propagation will be disabled on the output side for the initialization in the simplified initialization mode. |</p>
<table>
<thead>
<tr>
<th>Condition</th>
<th>Recommended Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Check for blocks that read input from conditionally executed subsystems during initialization.</td>
<td>Verify that the following behavior is acceptable. Some blocks, such as the Discrete-Time Integrator block, read their inputs from conditionally executed subsystems during initialization in the classic initialization mode. Simulink performs this step as an optimization technique. This optimization is not allowed in the simplified initialization mode because the output of a conditionally executed subsystem at the first time step after initialization may be different than the initial value declared in the corresponding Outport block. In particular, this discrepancy occurs if the subsystem is active at the first time step.</td>
</tr>
<tr>
<td>Check for a migration conflict for Outport blocks that use a Dialog as the <strong>Source of initial output value</strong>.</td>
<td>Other instances of Outport blocks with the same library link either cannot be migrated or are being migrated in a different manner. Review the results from the <strong>Check for library blocks with instances that cannot be migrated</strong> to learn about the different migration paths for other instances of each Outport block. The Outport block will maintain its current settings and use its specified <strong>Initial output</strong> value.</td>
</tr>
<tr>
<td>Condition</td>
<td>Recommended Action</td>
</tr>
<tr>
<td>---------------------------------------------------------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Check for a migration conflict for Outport blocks that use Input signal as the Source of initial output value.</td>
<td>Other instances of Outport blocks with the same library link either cannot be migrated or are being migrated in a different manner. Review the results from the <strong>Check for library blocks with instances that cannot be migrated</strong> to learn about the different migration paths for other instances of each Outport block.</td>
</tr>
<tr>
<td></td>
<td>The Outport block currently specifies an <strong>Initial output</strong> of [ ] (empty matrix), and the <strong>Output when disabled</strong> as held. This means that each outport does not perform initialization, but implicitly relies on source blocks to initialize its input signal.</td>
</tr>
<tr>
<td></td>
<td>After migration, the parameter <strong>Source of initial output value</strong> will be set to <strong>Input signal</strong> to reflect this behavior.</td>
</tr>
<tr>
<td>Check for a migration conflict for Outport blocks that have SimEvents semantics.</td>
<td>Other instances of Outport blocks with the same library link either cannot be migrated or are being migrated in a different manner. Review the results from the <strong>Check for library blocks with instances that cannot be migrated</strong> to learn about the different migration paths for other instances of each Outport block.</td>
</tr>
<tr>
<td></td>
<td>The Outport blocks will continue to use an <strong>Initial output</strong> value of [ ] (empty matrix) and an <strong>Output when disabled</strong> setting of held. Simulink will maintain these settings because their parent conditionally executed subsystems are connected to SimEvents blocks.</td>
</tr>
<tr>
<td>Condition</td>
<td>Recommended Action</td>
</tr>
<tr>
<td>---------------------------------------------------------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Check for a migration conflict for innermost Outport blocks with variable-size input and unspecified <strong>Initial output</strong>.</td>
<td>For these Outport blocks, the signal size varies only when the parent subsystem of the block is re-enabled. Therefore, Simulink implicitly assumes that the <strong>Initial output</strong> parameter is equal to 0, even though the parameter is unspecified, []. Consequently, unless you specify the parameter, the Model Advisor will explicitly set the parameter to 0 when the model is migrated to the simplified initialization mode. Other instances of Outport blocks with the same library link either cannot be migrated or are being migrated in a different manner. Review the results from the <strong>Check for library blocks with instances that cannot be migrated</strong> to learn about the different migration paths for other instances of each Outport block.</td>
</tr>
<tr>
<td>Check for a migration conflict for Outport blocks that use a default ground value as the Initial output.</td>
<td>The parameter <strong>Initial output</strong> is set to [] (empty matrix) and the source of the Outport is an invalid initial condition source. Thus, the block uses the default initial value as the initial output in the simplified initialization mode. Other instances of Outport blocks with the same library link either have errors or are being migrated differently.</td>
</tr>
<tr>
<td>Check for a migration conflict for merged Outport blocks without explicit specification of <strong>Initial output</strong>.</td>
<td>Review the results from the subcheck <strong>Check for library blocks with instances that cannot be migrated</strong> to learn about different migration paths for other instances of each Outport block. For the remaining Outport blocks, <strong>Initial output</strong> is set to [] (empty matrix) and <strong>Output when disabled</strong> is set to held respectively, in simplified initialization mode.</td>
</tr>
</tbody>
</table>
See Also

- “Migrating to Simplified Initialization Mode Overview” on page 10-5
- “Model Advisor Exclusion Overview” (Simulink Check)

**Check usage of Discrete-Time Integrator blocks**

**Check ID**: mathworks.design.DiscreteBlock

Identify Discrete-Time Integrator blocks with parameter settings that can lead to unexpected behavior, and help migrate your model to simplified initialization mode.

**Note** Run this check along with the other checks in the “Migrating to Simplified Initialization Mode Overview” on page 10-5.

**Description**

Simplified initialization mode was introduced in R2008b to improve the consistency of simulation results. For more information, see “Simplified Initialization Mode” and “Classic Initialization Mode”.

This Model Advisor check identifies settings in Discrete-Time Integrator blocks in your model that can cause problems if you use the simplified initialization mode. It also recommends settings for consistent behavior of Discrete-Time Integrator blocks. The results of the subchecks contain two types of statements: Failed and Warning. Failed statements identify issues that you must address manually before you can migrate the model to the simplified initialization mode. Warning statements identify issues or changes in behavior that can occur after migration.

**Results and Recommended Actions**

<table>
<thead>
<tr>
<th>Condition</th>
<th>Recommended Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Check for Discrete-Time Integrator blocks whose parameter <strong>Initial condition setting</strong> is set to <strong>Output</strong>.</td>
<td>Determine if the new behavior of the Discrete-Time Integrator blocks is acceptable. If it is not acceptable, modify the model to account for the new behavior before migrating to the simplified initialization mode.</td>
</tr>
</tbody>
</table>
### Condition

<table>
<thead>
<tr>
<th>Condition</th>
<th>Recommended Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Check for Discrete-Time Integrator blocks whose <strong>Initial condition setting</strong> parameter is set to State (most efficient) and are in a subsystem that uses triggered sample time.</td>
<td>Use periodic sample time for the block, or set <strong>Initial Condition setting</strong> to Output.</td>
</tr>
<tr>
<td>Check for blocks inside of the Iterator Subsystem that require elapsed time.</td>
<td>Within an Iterator Subsystem hierarchy, do not use blocks that require a service that maintains the time that has elapsed between two consecutive executions. Since an Iterator Subsystem can execute multiple times at a given time step, the concept of elapsed time is not well-defined between two such executions. Using these blocks inside of an Iterator Subsystem can cause unexpected behavior.</td>
</tr>
</tbody>
</table>

### See Also

- “Migrating to Simplified Initialization Mode Overview” on page 10-5
- “Model Advisor Exclusion Overview” (Simulink Check)

### Check model settings for migration to simplified initialization mode

**Note**  Do not run this check in isolation. Run this check along with the other checks in the “Migrating to Simplified Initialization Mode Overview” on page 10-5.

**Check ID:** `mathworks.design.ModelLevelMessages`

Identify settings in Model blocks and model configuration parameters that can lead to unexpected behavior, and help migrate your model to simplified initialization mode.
Description

Simplified initialization mode was introduced in R2008b to improve consistency of simulation results. For more information, see “Simplified Initialization Mode” and “Classic Initialization Mode”.

This Model Advisor check identifies issues in the model configuration parameters and Model blocks in your model that can cause problems when you migrate to simplified initialization mode. The results of the subchecks contain two types of statements: Failed and Warning. Failed statements identify issues that you must address manually before you can migrate the model to simplified initialization mode. Warning statements identify issues or changes in behavior that can occur after migration.

After running this Model Advisor consistency check, if you click Explore Result button, the messages pertain only to blocks that are not library-links.

Note Because it is difficult to undo these changes, select File > Save Restore Point As to back up your model before migrating to the simplified initialization mode.

For more information, see “Model Configuration Parameters: Connectivity Diagnostics”.

Results and Recommended Actions

<table>
<thead>
<tr>
<th>Condition</th>
<th>Recommended Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Verify that all Model blocks are using the simplified initialization mode.</td>
<td>Migrate the model referenced by the Model block to the simplified initialization mode, then migrate the top model.</td>
</tr>
<tr>
<td>Verify simplified initialization mode setting</td>
<td>Set Configuration Parameters &gt; Underspecified initialization detection to Simplified.</td>
</tr>
</tbody>
</table>

Action Results

Clicking Modify Settings causes the following:

- The Model parameter is set to simplified
- If an Outport block has the Initial output parameter set to the empty character vector, [], then the SourceOfInitialOutputValue parameter is set to Input signal.
• If an Outport has an empty **Initial output** and a variable-size signal, then the **Initial output** is set to zero.

**See Also**

• “Migrating to Simplified Initialization Mode Overview” on page 10-5
• “Model Advisor Exclusion Overview” (Simulink Check)

**Check S-functions in the model**

**Check ID**: mathworks.design.SFuncAnalyzer

Perform quality checks on S-functions in Simulink models or subsystems.

**Description**

The S-function analyzer performs quality checks on S-functions to identify improvements and potential problems in the specified model.

**Results and Recommended Actions**

<table>
<thead>
<tr>
<th>Condition</th>
<th>Recommended Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Continuous states are modified in mdlOutputs method.</td>
<td>Modify Continuous States at a major time step and use ssSetSolverNeedsReset function in S-function code.</td>
</tr>
<tr>
<td>Continuous states are modified in the mdlUpdate method.</td>
<td>Modify Continuous States only at a major time step and use ssSetSolverNeedsReset function in S-function code.</td>
</tr>
<tr>
<td>S-function discrete states are modified in the mdlOutputs at a minor step.</td>
<td>Modify the discrete states only at a major step guarded by ssIsMajorTimeStep function.</td>
</tr>
<tr>
<td>S-function mode vector is modified in the mdlOutputs method at a minor step.</td>
<td>Modify the mode vector only at a major step guarded by sslsMajorTimeStep function.</td>
</tr>
<tr>
<td>S-function is using static or global variables to declare internal states.</td>
<td>Declare the states explicitly using ssSetNumDiscStates function or “Model Global Data by Creating Data Stores”.</td>
</tr>
<tr>
<td>Condition</td>
<td>Recommended Action</td>
</tr>
<tr>
<td>--------------------------------------------------------------------------</td>
<td>-------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>S-function has continuous states but sample time is not declared continuous.</td>
<td>Specify continuous sample time using ssSetSampleTime function.</td>
</tr>
<tr>
<td>S-function has discrete states but the mdlOutputs and mdlUpdate methods are combined.</td>
<td>Define the mdlOutputs and mdlUpdate methods separately and modify discrete states only in mdlUpdate method.</td>
</tr>
<tr>
<td>S-function sets the SS_OPTION_CAN_BE_CALLED_CONDITIONALLY option when having state-like data or multiple sample times.</td>
<td>Remove the options when the S-function has state-like data or multiple sample times.</td>
</tr>
<tr>
<td>MEX compilers do not exist on the machine.</td>
<td>Check for the presence or install MEX compilers on the machine.</td>
</tr>
<tr>
<td>S-function encounters errors while compiling the model.</td>
<td>Check the Diagnostic Viewer output and recompile the model.</td>
</tr>
</tbody>
</table>

**Check for non-continuous signals driving derivative ports**

**Check ID:** mathworks.design.NonContSigDerivPort

Identify noncontinuous signals that drive derivative ports.

**Description**

Noncontinuous signals that drive derivative ports cause the solver to reset every time the signal changes value, which slows down simulation.

**Results and Recommended Actions**

<table>
<thead>
<tr>
<th>Condition</th>
<th>Recommended Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>There are noncontinuous signals in the model driving derivative ports.</td>
<td>• Make the specified signals continuous.</td>
</tr>
<tr>
<td></td>
<td>• Replace the continuous blocks receiving these signals with discrete state versions of the blocks.</td>
</tr>
</tbody>
</table>

**Capabilities and Limitations**

If you have a Simulink Check license, you can exclude blocks and charts from this check.
See Also

- “Simulink Models” on page 13-7
- “Simulation Phases in Dynamic Systems”
- “Model Advisor Exclusion Overview” (Simulink Check)

Runtime diagnostics for S-functions

Check ID: mathworks.design.DiagnosticSFcn

Check array bounds and solver consistency if S-Function blocks are in the model.

Description

Validates whether S-Function blocks adhere to the ODE solver consistency rules that Simulink applies to its built-in blocks.

Results and Recommended Actions

<table>
<thead>
<tr>
<th>Condition</th>
<th>Recommended Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Solver data inconsistency is set to none.</td>
<td>In the Configuration Parameters dialog box, set <strong>Solver data inconsistency</strong> to warning or error.</td>
</tr>
<tr>
<td>Array bounds exceeded is set to none.</td>
<td>In the Configuration Parameters dialog box, set <strong>Array bounds exceeded</strong> to warning or error.</td>
</tr>
</tbody>
</table>

Capabilities and Limitations

If you have a Simulink Check license, you can exclude blocks and charts from this check.

See Also

- “What Is an S-Function?”
- “How S-Functions Work”
- “Model Advisor Exclusion Overview” (Simulink Check)
Check model for foreign characters

Check ID: mathworks.design.characterEncoding

Check for characters that are incompatible with the current encoding

Description

Check for characters in the model file that cannot be represented in the current encoding. These can cause errors during simulation, and may be corrupted when saving the model.

Results and Recommended Actions

<table>
<thead>
<tr>
<th>Condition</th>
<th>Recommended Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Incompatible characters found</td>
<td>Change the current encoding to the encoding specified in the model file, using slCharacterEncoding. To change the current encoding you need to close the models, and this closes the Model Advisor.</td>
</tr>
</tbody>
</table>

Tips

The Upgrade Advisor report shows the encoding you need, or you can retrieve the encoding from the model using the command:

get_param(modelname,'SavedCharacterEncoding')

Use slCharacterEncoding to change the encoding. This setting applies to the current MATLAB session, so if you restart MATLAB and want to open the same model, you will need to make the same change to the current encoding again.

For more information see:

- slCharacterEncoding
- “Open a Model with Different Character Encoding”
- “Save Models with Different Character Encodings”

See Also

- “Consult the Upgrade Advisor”.
• “Model Upgrades”

**Identify unit mismatches in the model**

**Check ID:** mathworks.design.UnitMismatches

Identify instances of unit mismatches between ports in the model.

**Description**

Check for instances of unit mismatches between ports in the model.

**Results and Recommended Actions**

<table>
<thead>
<tr>
<th>Condition</th>
<th>Recommended Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unit mismatches found</td>
<td>Change one of the mismatched unit settings to match the unit settings for the other port.</td>
</tr>
</tbody>
</table>

**See Also**

• “Unit Specification in Simulink Models”.

**Identify automatic unit conversions in the model**

**Check ID:** mathworks.design.AutoUnitConversions

Identify instances of automatic unit conversions in the model.

**Description**

Identify instances of automatic unit conversions in the model.

**Results and Recommended Actions**

<table>
<thead>
<tr>
<th>Condition</th>
<th>Recommended Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Automatic unit conversions found</td>
<td>Check that the converted units are expected for the model.</td>
</tr>
</tbody>
</table>
See Also

- “Unit Specification in Simulink Models”.

**Identify disallowed unit systems in the model**

**Check ID:** mathworks.design.DisallowedUnitSystems

Identify instances of disallowed unit systems in the model.

**Description**

Identify instances of disallowed unit systems in the model.

**Results and Recommended Actions**

<table>
<thead>
<tr>
<th>Condition</th>
<th>Recommended Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Disallowed unit systems found</td>
<td>Either choose a unit that conforms to the configured unit system, or select another unit system. For more information, see “Restricting Unit Systems”.</td>
</tr>
</tbody>
</table>

See Also

- “Unit Specification in Simulink Models”.

**Identify undefined units in the model**

**Check ID:** mathworks.design.UndefinedUnits

Identify instances of unit specifications, not defined in the unit database, in the model.

**Description**

Identify instances of unit specifications, not defined in the unit database, in the model.
Results and Recommended Actions

<table>
<thead>
<tr>
<th>Condition</th>
<th>Recommended Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Undefined units found</td>
<td>Change the unit to one that Simulink supports.</td>
</tr>
</tbody>
</table>

See Also

- “Unit Specification in Simulink Models”.
- Allowed Units

Check model for block upgrade issues

Check ID: mathworks.design.Update

Check for common block upgrade issues.

Description

Check blocks in the model for compatibility issues resulting from using a new version of Simulink software.

Results and Recommended Actions

<table>
<thead>
<tr>
<th>Condition</th>
<th>Recommended Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blocks with compatibility issues found.</td>
<td>Click Modify to fix the detected block issues.</td>
</tr>
<tr>
<td>Check update status for the Level 2 API S-functions.</td>
<td>Consider replacing Level 1 S-functions with Level 2.</td>
</tr>
</tbody>
</table>

Action Results

Clicking Modify replaces blocks from a previous release of Simulink software with the latest versions.

See Also

- “Write Level-2 MATLAB S-Functions”.
- “Consult the Upgrade Advisor”.  

10-55
• “Model Upgrades”

Check model for block upgrade issues requiring compile time information

Check ID: mathworks.design.UpdateRequireCompile

Check for common block upgrade issues.

Description

Check blocks for compatibility issues resulting from upgrading to a new version of Simulink software. Some block upgrades require the collection of information or data when the model is in the compile mode. For this check, the model is set to compiled mode and then checked for upgrades.

Results and Recommended Actions

<table>
<thead>
<tr>
<th>Condition</th>
<th>Recommended Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model contains Lookup Table or Lookup Table (2-D) blocks and some of the blocks specify Use Input Nearest or Use Input Above for a lookup method.</td>
<td>Replace Lookup Table blocks and Lookup Table (2-D) blocks with n-D Lookup Table blocks. Do not apply Use Input Nearest or Use Input Above for lookup methods; select another option.</td>
</tr>
<tr>
<td>Model contains Lookup Table or Lookup Table (2-D) blocks and some blocks perform multiplication first during interpolation.</td>
<td>Replace Lookup Table blocks and Lookup Table (2-D) blocks with n-D Lookup Table blocks. However, because the n-D Lookup Table block performs division first, this replacement might cause a numerical difference in the result.</td>
</tr>
<tr>
<td>Model contains Lookup Table or Lookup Table (2-D) blocks. Some of these blocks specify Interpolation-Extrapolation as the Lookup method but their input and output are not the same floating-point type.</td>
<td>Replace Lookup Table blocks and Lookup Table (2-D) blocks with n-D Lookup Table blocks. Then change the extrapolation method or the port data types for block replacement.</td>
</tr>
<tr>
<td>Condition</td>
<td>Recommended Action</td>
</tr>
<tr>
<td>---------------------------------------------------------------------------</td>
<td>--------------------------------------------------------</td>
</tr>
<tr>
<td>Model contains Unit Delay blocks with <strong>Sample time</strong> set to -1 that inherit a continuous sample time.</td>
<td>Replace Unit Delay blocks with Memory blocks.</td>
</tr>
</tbody>
</table>

**Check Data Store Memory blocks for multitasking**

**Action Results**

Clicking **Modify** replaces blocks from a previous release of Simulink software with the latest versions.

**See Also**

- n-D Lookup Table
- Unit Delay
- “Consult the Upgrade Advisor”
- “Model Upgrades”

**Check that the model is saved in SLX format**

**Check ID:** `mathworks.design.UseSLXFile`

Check that the model is saved in SLX format.

**Description**

Check whether the model is saved in SLX format.

**Results and Recommended Actions**

<table>
<thead>
<tr>
<th>Condition</th>
<th>Recommended Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model not saved in SLX format</td>
<td>Consider upgrading to the SLX file format to use the latest features in Simulink.</td>
</tr>
</tbody>
</table>

**Capabilities and Limitations**

You can run this check on your library models.
**Tips**

Simulink Projects can help you upgrade models to SLX format and preserve file revision history in source control. See “Convert from MDL to SLX in a Project and Preserve Revision History” on page 15-677.

**See Also**

- “Save Models in the SLX File Format”
- “Consult the Upgrade Advisor”.
- “Model Upgrades”

**Check model for SB2SL blocks**

**Check ID:** mathworks.simulink.SB2SL.Check

Check that the model does not have outdated SB2SL blocks.

**Description**

Check if the model contains outdated SB2SL blocks.

**Results and Recommended Actions**

<table>
<thead>
<tr>
<th>Condition</th>
<th>Recommended Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model contains outdated SB2SL blocks</td>
<td>Consider upgrading the model to current SB2SL blocks.</td>
</tr>
</tbody>
</table>

**Action Results**

Clicking **Update SB2SL Blocks** replaces blocks with the latest versions.

**See Also**

- “Consult the Upgrade Advisor”.

**Check Model History properties**

**Check ID:** mathworks.design.SLXModelProperties

Check for edited model history properties
**Description**

Check models for edited Model History property values that could be used with source control tool keyword substitution. This keyword substitution is incompatible with SLX file format.

In the MDL file format you can configure some model properties to make use of source control tool keyword substitution. If you save your model in SLX format, source control tools cannot perform keyword substitution. Information in the model file from such keyword substitution is cached when you first save the MDL file as SLX, and is not updated again. The Model Properties History pane and Model Info blocks in your model show stale information from then on.

**Results and Recommended Actions**

<table>
<thead>
<tr>
<th>Condition</th>
<th>Recommended Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Edited model history properties</td>
<td>Manually or automatically reset the properties to the default values. Click the button to reset, or to inspect and change these properties manually, open the Model Properties dialog and look in the History pane.</td>
</tr>
</tbody>
</table>

**Capabilities and Limitations**

You can run this check on your library models.

**See Also**

- “Consult the Upgrade Advisor”.
- “Model Upgrades”

**Identify Model Info blocks that can interact with external source control tools**

**Check ID**: mathworks.design.ModelInfoKeywordSubstitution

Use this check to find Model Info blocks that can be altered by external source control tools through keyword substitution.
Description

This check searches for character vectors in the Model Info block enclosed within dollar signs that can be overwritten by an external source control tool. Using third-party source control tool keyword expansion might corrupt your model files when you submit them. Keyword substitution is not available in SLX model file format.

For a more flexible interface to source control tools, use a Simulink project instead of the Model Info block. See “About Source Control with Projects”.

Results and Recommended Actions

<table>
<thead>
<tr>
<th>Condition</th>
<th>Recommended Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>The Model Info block contains fields like this: $keyword$</td>
<td>Review the list of fields in the report, then remove the keyword character vectors from the Model Info block.</td>
</tr>
</tbody>
</table>

See Also

- “Consult the Upgrade Advisor”.
- “About Source Control with Projects”

Identify Model Info blocks that use the Configuration Manager

Check ID: mathworks.design.ModelInfoConfigurationManager

Use this check to find Model Info blocks that use the Configuration Manager.

Description

Model Info blocks using the Configuration Manager allow risky keyword substitution using external source control tools. Using third-party source control tool keyword expansion might corrupt your model files when you submit them. Keyword substitution is not available in SLX model file format. The Configuration Manager for the Model Info block will be removed in a future release.

For a more flexible interface to source control tools, use a Simulink project instead of the Model Info block. See “About Source Control with Projects”.

10-60
Results and Recommended Actions

<table>
<thead>
<tr>
<th>Condition</th>
<th>Recommended Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>A Model Info block is using the Configuration Manager.</td>
<td>Click Remove the Configuration Manager.</td>
</tr>
</tbody>
</table>

See Also

- “Consult the Upgrade Advisor”.
- “About Source Control with Projects”

Check model for legacy 3DoF or 6DoF blocks

Check ID: mathworks.design.Aeroblks.CheckDOF

Lists 3DoF and 6DoF blocks are outdated.

Description

This check searches for 3DoF and 6DoF blocks from library versions prior to 3.13 (R2014a).

Results and Recommended Actions

<table>
<thead>
<tr>
<th>Condition</th>
<th>Recommended Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blocks configured with old versions of 3DoF or 6DoF blocks found.</td>
<td>Click Replace 3DoF and 6DoF Blocks to replace the blocks with latest versions.</td>
</tr>
</tbody>
</table>

Action Results

Clicking Replace 3DoF and 6DoF Blocks replaces blocks with the latest versions.

See Also

- “Equations of Motion” (Aerospace Blockset)
Check model and local libraries for legacy Aerospace Blockset blocks

Check ID: mathworks.design.Aeroblks.CheckFG

Lists blocks configured to use FlightGear versions that are outdated or not supported.

Description

This check searches and lists blocks configured to use FlightGear versions that are outdated or not supported.

Results and Recommended Actions

<table>
<thead>
<tr>
<th>Condition</th>
<th>Recommended Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blocks configured with old</td>
<td>Click <strong>Update FlightGear blocks</strong> to change block settings to latest supported</td>
</tr>
<tr>
<td>versions of FlightGear are found.</td>
<td>version of FlightGear. Then, download latest version of FlightGear that MATLAB</td>
</tr>
<tr>
<td></td>
<td>supports.</td>
</tr>
</tbody>
</table>

Action Results

Clicking **Update FlightGear blocks** changes block settings to the latest supported version of FlightGear.

See Also

- “Flight Simulator Interfaces” (Aerospace Blockset)

Check model for Aerospace Blockset navigation blocks

Check ID: mathworks.design.Aeroblks.CheckNAV

Searches for Three-Axis Inertial Measurement Unit, Three-Axis Gyroscope, and Three-Axis Accelerometer blocks prior to 3.21 (R2018a).

Description

This check searches for Three-Axis Inertial Measurement Unit, Three-Axis Gyroscope, and Three-Axis Accelerometer blocks that have been updated in R2018a.
### Results and Recommended Actions

<table>
<thead>
<tr>
<th>Condition</th>
<th>Recommended Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Three-Axis Inertial Measurement Unit, Three-Axis Gyroscope, and Three-Axis Accelerometer blocks prior to R2018a.</td>
<td>In R2018a or later, if you did not previously solve for steady state conditions, save the model now. If you previously solved for steady state conditions for the model, solve for these steady state conditions again, and then save the model.</td>
</tr>
</tbody>
</table>

#### See Also
- Three-Axis Accelerometer
- Three-Axis Gyroscope
- Three-Axis Inertial Measurement Unit

#### Check and update masked blocks in library to use promoted parameters

**Check ID:** mathworks.design.CheckAndUpdateOldMaskedBuiltinBlocks

Check for libraries that should be updated to use promoted parameters.

**Description**

This check searches libraries created before R2011b for masked blocks that should be updated to use promoted parameters. Since R2011b, if a block parameter is not promoted, its value in the linked block is locked to its value in the library block. This check excludes blocks of type Subsystem, Model reference, S-Function and M-S-Function.

#### Results and Recommended Actions

<table>
<thead>
<tr>
<th>Condition</th>
<th>Recommended Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Libraries that need to be updated are found</td>
<td>Click <strong>Update</strong>. Once the libraries have been updated, run the check again</td>
</tr>
</tbody>
</table>

#### Capabilities and Limitations

You can:
• Run this check on your library models.
• Exclude blocks and charts from this check if you have a Simulink Check license.

See Also
• “Consult the Upgrade Advisor”.
• “Model Upgrades”
• “Model Advisor Exclusion Overview” (Simulink Check)

Check and update mask image display commands with unnecessary imread() function calls

Check ID: mathworks.design.CheckMaskDisplayImageFormat

Check identifies masks using image display commands with unnecessary calls to the imread() function.

Description
This check searches for the mask display commands that make unnecessary calls to the imread() function, and updates them with mask display commands that do not call the imread() function. Since 2013a, a performance and memory optimization is available for mask images specified using the image path instead of the RGB triple matrix.

Results and Recommended Actions

<table>
<thead>
<tr>
<th>Condition</th>
<th>Recommended Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mask display commands that make unnecessary calls to the imread() function are found.</td>
<td>Click Update. Once the blocks have been updated, run the check again.</td>
</tr>
</tbody>
</table>

Capabilities and Limitations

You can:
• Run this check on your library models.
• Exclude blocks and charts from this check if you have a Simulink Check license.
See Also

• “Consult the Upgrade Advisor”.
• “Model Upgrades”
• “Model Advisor Exclusion Overview” (Simulink Check)

Check and update mask to affirm icon drawing commands
dependency on mask workspace

Check ID: mathworks.design.CheckMaskRunInitFlag

Check identifies if the mask icon drawing commands have dependency on the mask workspace.

Description

This check identifies if the mask icon drawing commands have dependency on the mask workspace and updates the RunInitForIconRedraw property accordingly. If there is no mask workspace dependency, the value of RunInitForIconRedraw is set to off, whereas, if there is mask workspace dependency the values is set to on.

Setting the values of RunInitForIconRedraw to off when there is no mask workspace dependency optimizes the performance by not executing the mask initialization code before drawing the block icon.

Results and Recommended Actions

<table>
<thead>
<tr>
<th>Condition</th>
<th>Recommended Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mask drawing commands that are dependent or independent of the mask workspace are found.</td>
<td>Click Update. Once the blocks have been updated, run the check again.</td>
</tr>
</tbody>
</table>

Capabilities and Limitations

You can:

• Run this check on your library models.
• Exclude blocks and charts from this check if you have a Simulink Check license.
See Also

- “Consult the Upgrade Advisor”.
- “Model Upgrades”
- “Model Advisor Exclusion Overview” (Simulink Check)

**Identify masked blocks that specify tabs in mask dialog using MaskTabNames parameter**

**Check ID:** mathworks.design.CheckAndUpdateOldMaskTabNames

This check identifies masked blocks that specify tabs in mask dialog using the MaskTabNames parameter.

**Description**

This check identifies masked blocks that use the MaskTabNames parameter to programmatically create tabs in the mask dialog. Since R2013b, dialog controls are used to group parameters in a tab on the mask dialog.

**Results and Recommended Actions**

<table>
<thead>
<tr>
<th>Condition</th>
<th>Recommended Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Masked blocks that use the MaskTabNames parameter to create tabs programmatically in the mask dialog are found.</td>
<td>Click <strong>Upgrade</strong> available in the <strong>Action</strong> section. Once the blocks have been updated, run the check again.</td>
</tr>
</tbody>
</table>

**Capabilities and Limitations**

You can run this check on your library models.

**See Also**

- “Consult the Upgrade Advisor”.
- “Model Upgrades”
Identify questionable operations for strict single-precision design

Check ID: mathworks.design.StowawayDoubles

For a strict single-precision design, this check identifies the blocks that introduce double-precision operations, and non-optimal model settings.

Description

For a strict single-precision design, this check identifies the blocks that introduce double-precision operations, and non-optimal model settings.

Results and Recommended Actions

<table>
<thead>
<tr>
<th>Condition</th>
<th>Recommended Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Double-precision floating-point operations found in model.</td>
<td>Verify that:</td>
</tr>
<tr>
<td></td>
<td>• Block input and output data types are set correctly.</td>
</tr>
<tr>
<td></td>
<td>• In the Configuration Parameters dialog box, <strong>Default for underspecified data type</strong> is set to <strong>single</strong>.</td>
</tr>
<tr>
<td>Model uses a library standard that is not optimal for strict-single designs.</td>
<td>Verify that:</td>
</tr>
<tr>
<td></td>
<td>• All target-specific math libraries used by the model support single-precision implementations.</td>
</tr>
<tr>
<td></td>
<td>Set <strong>Configuration Parameters &gt; Standard math library</strong> to <strong>C99 (ISO)</strong>.</td>
</tr>
<tr>
<td>Condition</td>
<td>Recommended Action</td>
</tr>
<tr>
<td>----------------------------------------------------</td>
<td>--------------------------------------------------------</td>
</tr>
<tr>
<td>Logic signals are not implemented as Boolean data.</td>
<td>Verify that:</td>
</tr>
<tr>
<td></td>
<td>• In the Configuration Parameters dialog box, <strong>Implement logic signals as Boolean data</strong> is selected.</td>
</tr>
</tbody>
</table>

**Capabilities and Limitations**

If you have a Simulink Check license, you can exclude blocks and charts from this check.

**See Also**

• “Validate a Floating-Point Embedded Model”
• “Consult the Upgrade Advisor”.
• “Model Upgrades”
• “Model Advisor Exclusion Overview” (Simulink Check)

**Check get_param calls for block CompiledSampleTime**

**Check ID:** mathworks.design.CallsGetParamCompiledSampleTime

Use this check to identify MATLAB files in your working environment that contain get_param function calls to return the block `CompiledSampleTime` parameter.

**Description**

For multi-rate blocks (including subsystems), Simulink returns the block compiled sample time as a cell array of the sample rates in the block. The return value is a cell array of pairs of doubles. MATLAB code that accepts this return value only as pairs of doubles can return an error when called with a multi-rate block. Use this check to identify such code in your environment. Modify these instances of code to accept a cell array of pairs of doubles instead.

For example, consider a variable `blkTs`, which has been assigned the compiled sample time of a multi-rate block.

```matlab
blkTs = get_param(block,'CompiledSampleTime');
```
Here are some examples in which the original code works only if blkTs is a pair of doubles and the block is a single-rate block:

• Example 1

```matlab
if isinf(blkTs(1))
    disp('found constant sample time')
end
```

Since blkTs is now a cell array, Simulink gives this error message:

Undefined function 'isinf' for input arguments of type 'cell'

Instead, use this code, for which blkTs can be a cell array or a pair of doubles.

```matlab
if isequal(blkTs, [inf,0])
    disp('found constant sample time')
end
```

• Example 2

```matlab
if all(blkTs == [-1,-1])
    disp('found triggered sample time')
end
```

For the above example, since blkTs is now a cell array, Simulink gives this error:

Undefined function 'eq' for input arguments of type 'cell'

Instead, use this code, for which blkTs can be a cell array or a pair of doubles.

```matlab
if isequal(blkTs, [-1,-1])
    disp('found triggered sample time')
end
```

• Example 3

```matlab
if (blkTs(1) == -1)
    disp('found a triggered context')
end
```

Again, since blkTs is now a cell array, Simulink gives this error:

Undefined function 'eq' for input arguments of type 'cell'

Instead, use this code.
if ~iscell(blkTs)
    blkTs = {blkTs};
end
for idx = 1:length(blkTs)
    thisTs = blkTs{idx};
    if (thisTs(1) == -1)
        disp('found a triggered context')
    end
end

The above code checks for a triggered type sample time (triggered or async). In cases in which a block has constant sample time ([inf,0]) in addition to triggered or async or when a block has multiple async rates, this alternative property detects the triggered type sample time.

This check scans MATLAB files in your environment. If the check finds instances of MATLAB code that contain `get_param` calls to output the block compiled sample time, Upgrade Advisor displays these results. It suggests that you modify code that accepts the block compiled sample time from multi-rate blocks.

### Results and Recommended Actions

<table>
<thead>
<tr>
<th>Condition</th>
<th>Recommended Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>No MATLAB files call <code>get_param(block,CompiledSampleTime)</code></td>
<td>None</td>
</tr>
<tr>
<td>Some MATLAB files call <code>get_param(block,CompiledSampleTime)</code></td>
<td>If files use the block CompiledSampleTime parameter from multi-rate blocks, modify these files to accept the parameter as a cell array of pairs of doubles</td>
</tr>
</tbody>
</table>

### See Also

- “Sample Times in Subsystems”
- “Block Compiled Sample Time”

### Check if all simulation outputs are returned as a single Simulink.SimulationOutput object

**Check ID:** `mathworks.design.CheckSingleSimulationOutput`
Use this check to identify if the simulation result is returned as a single Simulink.SimulationOutput object.

**Description**

This check scans your model to verify if the parameter ReturnWorkspaceOutputs is enabled. Enabling this parameter, returns simulation outputs in a Simulink.SimulationOutput when simulating interactively. Simulation outputs include signal, state, output, DSM logging, and scope and To Workspace block logging.

When ReturnWorkspaceOutputs is enabled, it provides:

- Easier management of simulation data.
- Automatic access to SimulationMetadata.
- Compatibility with multiple parallel simulations and batch simulations.

**Results and Recommended Actions**

<table>
<thead>
<tr>
<th>Condition</th>
<th>Recommended Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simulation outputs are not returned as a single Simulink.SimulationOutput object</td>
<td>Return all simulation outputs as a single Simulink.SimulationOutput object</td>
</tr>
</tbody>
</table>

**See Also**

- “Single simulation output”
- Simulink.SimulationOutput

**Check model for parameter initialization and tuning issues**

**Check ID**: mathworks.design.ParameterTuning

Use this check to identify issues in the model that occur when you initialize parameters or tune them.

**Description**

This check scans your model for parameter initialization and tuning issues like:

- Rate mismatch between blocks
• Divide by zero issue in conditionally executed subsystems
• Invalid control port value in Index Vector blocks

Results and Recommended Actions

<table>
<thead>
<tr>
<th>Condition</th>
<th>Recommended Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>The model has rate transition issues.</td>
<td>Select <strong>Automatically handle rate transition for data transfer</strong> in the <strong>Solver</strong> pane of the model configuration parameters.</td>
</tr>
<tr>
<td>The model has a divide by zero issue in a conditionally executed subsystem with a control port.</td>
<td>At the command prompt, run <code>set_param(control_port,'DisallowConstTsAndPrmTs','on')</code></td>
</tr>
<tr>
<td>The model has an invalid control port value in a conditionally executed subsystem.</td>
<td>At the command prompt, run <code>set_param(control_port,'DisallowConstTsAndPrmTs','on')</code></td>
</tr>
</tbody>
</table>

Action Results

Select **Upgrade model** to resolve issues in the model related to parameter initialization and tuning.

See Also

• “Automatic Rate Transition”

Check for virtual bus across model reference boundaries

Check ID: `mathworks.design.CheckVirtualBusAcrossModelReference`

Check virtual bus signals that cross model reference boundaries.

Description

This check identifies root-level Inport and Outport blocks in referenced models and Model blocks with virtual bus outputs that require updates to change to nonvirtual bus signals.

If the check identifies issues, click **Update Model** to convert root-level Inport and Outport blocks configured for virtual buses to use nonvirtual buses in these situations:
• For root-level Inport blocks — Enable the **Output as nonvirtual bus** parameter and insert a Signal Conversion block after the Inport block. The Signal Conversion block is configured to output a virtual bus.

• For root-level Outport blocks — Enable the **Output as nonvirtual bus in parent model** parameter.

• For Model blocks — For ports whose Outport blocks were updated to address issues, insert a Signal Conversion block after the corresponding ports of the Model block. The Signal Conversion block is configured to output a virtual bus.

**Recommended Action and Results**

To resolve issues, click **Upgrade Model**.

**Note** Run the **Analyze model hierarchy and continue upgrade sequence** check on the top-level model and then down through the model reference hierarchy.

Clicking **Upgrade Model** converts affected root-level Inport and Outport blocks configured for virtual buses to use nonvirtual buses in models where you:

• Use function prototype control
• Perform C++ code generation with the I/O arguments step method option.

Alternatively, you can change the C++ code generation function specification setting to **Default step method**:

1. In the **Configuration Parameters > Code Generation > Interface** pane, click **Configure C++ Class Interface**.
2. In the dialog box, set the **Function specification** parameter to **Default step method**.

• Use buses that have variable-dimension signals
• Use an associated non-auto storage class for Outport block signals

• The conversion for non-auto storage class occurs only if you have the target generation license that the model requires. For example, an ERT target requires an Embedded Coder license.

• Use Export-function models where an Outport block is driven by a nonvirtual bus

• Have Model blocks that reference models containing Outport blocks that have been fixed — Clicking **Upgrade Model** updates Model blocks referencing the models that
had Outport blocks fixed by the **Analyze model hierarchy and continue upgrade sequence** check.

**See Also**

- “Bus Data Crossing Model Reference Boundaries”

## Check model for custom library blocks that rely on frame status of the signal

**Check ID:** mathworks.design.DSPFrameUpgrade

This check identifies custom library blocks in the model that depend on the frame status of the signal.

**Description**

This check searches for the custom library blocks in a model that depend on the frame status of the signal. The check analyzes the blocks, recommends fixes, and gives reasons for the fixes. You must make the fixes manually.

### Results and Recommended Actions

<table>
<thead>
<tr>
<th>Condition</th>
<th>Recommended Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>The check finds custom library blocks that depend on the frame status of the signal.</td>
<td>Follow the recommendation given by the Upgrade Advisor.</td>
</tr>
</tbody>
</table>

**Capabilities and Limitations**

You can run this check only on custom library blocks in your model.

You must make the fixes manually.

This check appears only if you have the DSP System Toolbox installed.

**See Also**

“Frame-based processing” (DSP System Toolbox)
Check model for S-function upgrade issues

**Check ID:** 'mathworks.design.CheckForSFcnUpgradeIssues'

Use this check on your model to identify your S-function's upgrade compatibility issues. These issues may include the use of 32-bit APIs, compilation with incompatible options, or use of deprecated separate complex APIs. Some common issues and information related to the fixes are described in results and recommendations section below.

**Description**

When upgrading your S-functions to use the features in the latest release, this check scans your model to warn against S-function upgrade incompatibility issues. If the result of this check gives a warning or error, fix your C MEX S-functions according to the description.

**Results and Recommended Actions**

<table>
<thead>
<tr>
<th>Condition</th>
<th>Recommended Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Custom-built S-functions are not supported.</td>
<td>Recompile your S-function with available compatible options. See “Custom-built MEX File Not Supported In Current Release” (MATLAB) for more information.</td>
</tr>
<tr>
<td>S-function is not compiled with the latest API (mex -R2018a).</td>
<td>Recompile using the latest flag (mex -R2018a). See “MEX File Is Compiled With Outdated Option” (MATLAB) for more information.</td>
</tr>
<tr>
<td>S-function uses 32-bit functions.</td>
<td>Modify your code according to the instructions in “MEX File Calls A 32-bit Function” (MATLAB).</td>
</tr>
<tr>
<td>S-function is using deprecated separate complex APIs (mxGetPi, mxSetPi,</td>
<td>Use interleaved complex APIs and recompile your code with the latest flag (mex -R2018a). See “Upgrade MEX Files to Use Interleaved Complex API” (MATLAB) for more information.</td>
</tr>
<tr>
<td>mxGetImagData, mxSetImagData).</td>
<td></td>
</tr>
</tbody>
</table>
### See Also

- “MATLAB Data in C S-Functions”

### Check Rapid accelerator signal logging

**Check ID:** `mathworks.design.CheckRapidAcceleratorSignalLogging`

When simulating your model in rapid accelerator mode, use this check to find signals logged in your model that are globally disabled. Rapid accelerator mode supports signal logging. Use this check to enable signal logging globally.

#### Description

This check scans your model to see if a simulation is in rapid accelerator mode and whether the model contains signals with signal logging. If the check finds an instance and signal logging is globally disabled, an option to turn on signal logging globally appears.

#### Results and Recommended Actions

<table>
<thead>
<tr>
<th>Condition</th>
<th>Recommended Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simulation mode is not rapid accelerator.</td>
<td>None. You can enable signal logging in rapid accelerator mode.</td>
</tr>
<tr>
<td>Simulation mode is rapid accelerator. Upgrade Advisor did not find signals with signal logging enabled.</td>
<td>None. The model does not use signal logging. Enable signal logging for signals and globally if you want to log signals.</td>
</tr>
</tbody>
</table>
### Condition

<table>
<thead>
<tr>
<th>Condition</th>
<th>Recommended Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simulation mode is rapid accelerator. Upgrade Advisor found signals with</td>
<td>Enable signal logging globally if you want to log signals with signal logging</td>
</tr>
<tr>
<td>signal logging enabled. However, global setting for signal logging was</td>
<td>enabled.</td>
</tr>
<tr>
<td>disabled.</td>
<td></td>
</tr>
<tr>
<td>Signal logging was already globally enabled.</td>
<td>None.</td>
</tr>
</tbody>
</table>

#### Action Results

Selecting **Modify** enables signal logging globally in your model.

#### See Also

- “Signal Logging in Rapid Accelerator Mode”
- “Consult the Upgrade Advisor”.

### Check virtual bus inputs to blocks

**Check ID:** `mathworks.design.VirtualBusUsage`

Check bus input signals for a set of blocks.

#### Description

Check bus input signals for a set of blocks.

Starting in R2015b, virtual bus signal inputs to blocks that require nonbus or nonvirtual bus input can cause an error. Examples of blocks that can specify a bus object as their output data type include a Bus Creator block and a root Inport block. The blocks that cause an error when they have a virtual bus input in this situation are:

- Assignment
- Delay

The Delay block causes an error only if you use the Block Parameters dialog box to:

- Set an initial condition that is a MATLAB structure or zero.
- Specify a value for **State name**.
The Unit Delay block causes an error only if you use the Block Parameters dialog box to:
- Set an initial condition that is a MATLAB structure or zero.
- Specify a value for **State name**.
- Vector Concatenate

### Results and Recommended Actions

<table>
<thead>
<tr>
<th>Condition</th>
<th>Recommended Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Virtual bus signal input to these blocks:</td>
<td>In the Upgrade Advisor, click <strong>Modify</strong>.</td>
</tr>
<tr>
<td>• Assignment</td>
<td>The check inserts a Bus to Vector block to attempt to convert virtual bus input signals to vector signals. For issues that the Upgrade Advisor identifies but cannot fix, modify the model manually. For details, see “Correct Buses Used as Vectors”.</td>
</tr>
<tr>
<td>• Delay (if you specify an initial condition from the dialog box that is a MATLAB structure or zero and the value for <strong>State name</strong> is not empty)</td>
<td></td>
</tr>
<tr>
<td>• Permute Dimension</td>
<td></td>
</tr>
<tr>
<td>• Reshape</td>
<td></td>
</tr>
<tr>
<td>• Selector</td>
<td></td>
</tr>
<tr>
<td>• Unit Delay (if you specify an initial condition that is a MATLAB structure or zero and the value for <strong>State name</strong> is not empty)</td>
<td></td>
</tr>
<tr>
<td>• Vector Concatenate</td>
<td></td>
</tr>
</tbody>
</table>

### Action Results

Clicking **Modify** inserts a Bus to Vector block at the input ports of blocks.

For many models, running the Upgrade Advisor modifies your model so that bus signals are not treated as vectors. However, for some models you can encounter compatibility issues even after running the check. Modify your model manually to address those issues.
After you compile the model using Upgrade Advisor, the Simulink Editor sometimes indicates that you need to save the model (the model is dirty), even though you did not make changes. To prevent this issue from reoccurring for this model, save the model.

<table>
<thead>
<tr>
<th>Modeling Pattern</th>
<th>Issue</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Store Memory block with <strong>Data Type</strong> set to <strong>Inherit: auto</strong></td>
<td>A Data Store Memory block whose associated Data Store Read or Data Store Write blocks read or write bus signal data must use a bus object.</td>
<td>In the Data Store Memory block, set the <strong>Data Type</strong> signal attribute to <strong>Bus: &lt;BusObject&gt;</strong>.</td>
</tr>
<tr>
<td>Signal Conversion block <strong>Output</strong> parameter matches input bus type</td>
<td>A Signal Conversion block whose <strong>Output</strong> parameter is set to <strong>Nonvirtual bus</strong> requires a virtual bus input. A Signal Conversion block whose <strong>Output</strong> parameter is set to <strong>Virtual bus</strong> requires a nonvirtual bus input.</td>
<td>To create a copy of the input signal, set <strong>Output</strong> to <strong>Signal copy</strong>.</td>
</tr>
<tr>
<td>Merge, Switch, or Multiport Switch block with multiple bus inputs</td>
<td>Merge, Switch, or Multiport Switch blocks with multiple bus inputs require those inputs to have the same names and hierarchy.</td>
<td>Reconfigure the model so that the bus inputs have the same names and hierarchy.</td>
</tr>
<tr>
<td>Root Inport block outputting a virtual bus and specifying a value for <strong>Port dimensions</strong></td>
<td>A root Inport block that outputs to a virtual bus must inherit the dimensions.</td>
<td>Set the Inport block <strong>Port dimensions</strong> signal attribute to 1 or -1 (inherit).</td>
</tr>
<tr>
<td>Mux block with nonvirtual bus inputs</td>
<td>A Mux block cannot accept nonvirtual bus signals.</td>
<td>To treat the output as an array, replace the Mux block with a Vector Concatenate block. If you want a virtual bus output, use a Bus Creator block to combine the signals.</td>
</tr>
<tr>
<td>Modeling Pattern</td>
<td>Issue</td>
<td>Solution</td>
</tr>
<tr>
<td>------------------</td>
<td>----------------------------------------------------------------------</td>
<td>-------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Bus to Vector block without a virtual bus signal input</td>
<td>A nonbus signal does not need a Bus to Vector block.</td>
<td>Remove the Bus to Vector block.</td>
</tr>
<tr>
<td>Assignment block with virtual bus inputs</td>
<td>The Upgrade Advisor converts the Assignment block Y0 port bus input to a vector.</td>
<td>Add a Bus to Vector block before the Assignment block.</td>
</tr>
<tr>
<td>S-function using a nonvirtual bus</td>
<td>An S-function that is not a Level-2 C S-function does not support nonvirtual bus signals.</td>
<td>Change the S-function to be a Level-2 C S-function.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Consider using an S-Function Builder block to create a Level-2 C S-function.</td>
</tr>
<tr>
<td>Stateflow chart with parameterized data type</td>
<td>In a Stateflow chart, you cannot parameterize the data type of an input or output in terms of another input or output if the data type is a bus object.</td>
<td>For the parameterized port, set <strong>Data Type</strong> to Bus: <code>&lt;object name&gt;</code>.</td>
</tr>
<tr>
<td>Subsystem with bus operations in a Stateflow chart</td>
<td>An Inport block inside a subsystem in a Stateflow chart requires a bus object data type if its signal is a bus.</td>
<td>In the Inport block, set <strong>Data type</strong> to Bus: <code>&lt;object name&gt;</code>.</td>
</tr>
<tr>
<td>Ground block used as a bus source</td>
<td>The output signal of a Ground block cannot be a source for a bus.</td>
<td>Use a Constant block with <strong>Constant value</strong> set to 0 and the <strong>Output data type</strong> signal attribute set to Bus: <code>&lt;object name&gt;</code>.</td>
</tr>
<tr>
<td>Root Outport block with a single-element bus object data type</td>
<td>The input to the Outport block must be a bus if it specifies a bus object as its data type.</td>
<td>In the Outport block, set <strong>Data type</strong> to <strong>Inherit: auto</strong>.</td>
</tr>
</tbody>
</table>

**See Also**

- Bus to Vector block
- “Correct Buses Used as Vectors”
- “Migrating to Simplified Initialization Mode Overview” on page 10-5
Check for root outports with constant sample time

Check ID: `mathworks.design.CheckConstRootOutportWithInterfaceUpgrade`

Use this check to identify root outports with a constant sample time used with an AUTOSAR target, Function Prototype Control, or the model C++ class interface.

Description

Root outports with constant sample time are not supported when using an AUTOSAR target, Function Prototype Control, or the model C++ class interface. Use this check to identify root Outport blocks with this condition and modify the blocks as recommended.

Results and Recommended Actions

<table>
<thead>
<tr>
<th>Condition</th>
<th>Recommended Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Root outport with constant sample time used with an AUTOSAR target, Function Prototype Control or the model C++ class interface.</td>
<td>Consider one of the following:</td>
</tr>
<tr>
<td></td>
<td>• Set the sample time of the block to the fundamental sample time.</td>
</tr>
<tr>
<td></td>
<td>• Identify the source of the constant sample time and set its sample time to the fundamental sample time.</td>
</tr>
<tr>
<td></td>
<td>• Place a Rate Transition block with inherited sample time (-1) before the block.</td>
</tr>
</tbody>
</table>

See Also

- “Consult the Upgrade Advisor”.

Analyze model hierarchy and continue upgrade sequence


Check for child models and guide you through upgrade checks.
Description

This check identifies child models of this model, and guides you through upgrade checks to run both non-compile and compile checks. The Advisor provides tools to help with these tasks:

- If the check finds child models, it offers to run the Upgrade Advisor upon each child model in turn and continue the upgrade sequence. If you have a model hierarchy you need to check and update each child model in turn.
- If there are no child models, you still need to continue the check sequence until you have run both non-compile and compile checks.

You must run upgrade checks in this order: first the checks that do not require compile time information and do not trigger an Update Diagram, then the compile checks.

Click **Continue Upgrade Sequence** to run the next checks. If there are child models, this will open the next model. Keep clicking **Continue Upgrade Sequence** until the check passes.

Results and Recommended Actions

<table>
<thead>
<tr>
<th>Condition</th>
<th>Recommended Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Child models found</td>
<td>Click <strong>Continue Upgrade Sequence</strong> to run the next checks. If there are child models, this will close the current Upgrade Advisor session, and open Upgrade Advisor for the next model in the hierarchy.</td>
</tr>
<tr>
<td>No child models, but more checks to run</td>
<td>If there are no child models, click <strong>Continue Upgrade Sequence</strong> to refresh the Upgrade Advisor with compilation checks selected. The compile checks trigger an Update Diagram (marked with ^). Run the next checks and take advised actions. When you return to this check, click <strong>Continue Upgrade Sequence</strong> until this check passes.</td>
</tr>
</tbody>
</table>
**Tips**

Best practice for upgrading a model hierarchy is to check and upgrade each model starting at the leaf end and working up to the root model.

When you click **Continue Upgrade Sequence**, the Upgrade Advisor opens the leaf model as far inside the hierarchy as it can find. Subsequent steps guide you through upgrading your hierarchy from leaf to root model.

When you open the Upgrade Advisor, the checks that are selected do not require compile time information and do not trigger an Update Diagram. Checks that trigger an Update Diagram are not selected to run by default, and are marked with ^. When you use the Upgrade Advisor on a hierarchy, keep clicking **Continue Upgrade Sequence** to move through this sequence of analysis:

1. The Upgrade Advisor opens each model and library in turn, from leaf to root, and selects the non-compile checks. Run the checks, take any advised actions, then click **Continue Upgrade Sequence** to open the next model and continue.

2. When you reach the root end of the hierarchy, the Upgrade Advisor then opens each model again in the same order (but not libraries) and selects only the checks that require a model compile. Run the checks, take any advised actions, then click **Continue Upgrade Sequence** to open the next model. Continue until you reach the end of the hierarchy and this check passes.

**See Also**

- “Consult the Upgrade Advisor”.
- “Model Upgrades”

**Check Access to Data Stores**

**Check ID:** mathworks.design.ConflictsForDataStoreReadWriters

Identify potential execution order sensitivity when reading and writing to data stores.

**Description**

The execution order of blocks that read and write to the same data store can change the simulation result. When blocks in the same hierarchy access the same data store, the execution order is not deterministic.
## Results and Recommended Actions

<table>
<thead>
<tr>
<th>Condition</th>
<th>Recommended Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Store Memory block accessed by multiple blocks in the same hierarchy.</td>
<td>To enforce execution order for the blocks, consider the following: • Add a data dependency between the blocks. • Set block priority. • Move blocks into separate Function-Call Subsystem blocks and schedule them.</td>
</tr>
</tbody>
</table>

### See Also

- “Local and Global Data Stores”
- Data Store Memory
- Data Store Read
- Data Store Write

### Check relative execution orders for Data Store Read and Data Store Write blocks

**Check ID:** mathworks.design.TaskBasedSorting

Check relative execution order changes between legacy and task-based sorting for Data Store Read and Data Store Write blocks.

**Description**

Legacy models used block sorting to determine block execution order. With task-based sorting, the relative execution order involving Data Store Memory blocks can change. This check detects the changes and provides an option to update your model with the original execution order.
Results and Recommended Actions

<table>
<thead>
<tr>
<th>Condition</th>
<th>Recommended Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution order with a Data Store Memory block changes. You want the original execution order.</td>
<td>Click the <strong>Modify</strong> button.</td>
</tr>
</tbody>
</table>

See Also

- “Task-Based Sorting to Determine Block Execution Order”
Model Reference Conversion
Advisor
Model Reference Conversion Advisor

Check Conversion Input Parameters

Use input parameters to configure the actions the advisor performs and the output it produces.

You can use the default parameters to run the advisor without changing any parameters.

<table>
<thead>
<tr>
<th>Input Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>New model name</td>
<td>The advisor provides a model name that is based on the Subsystem block name and is unique in the MATLAB path. The model name cannot exceed 59 characters.</td>
</tr>
<tr>
<td></td>
<td><strong>Tip</strong> If the advisor generates an error indicating that the target referenced model already exists, then use the <strong>New model name</strong> parameter to specify a new file name.</td>
</tr>
<tr>
<td>Conversion data file name</td>
<td>The advisor creates a file for storing data created during the conversion. By default, the advisor uses the model name at the beginning of the file name and appending _conversion_data.mat. For example, if the subsystem is in a model named myModel, the conversion file name is myModel_conversion_data.mat. You can save the conversion data in a MAT-file (default) or a MATLAB file. If you use a .m file extension, the advisor serializes all variables to a MATLAB file.</td>
</tr>
<tr>
<td></td>
<td><strong>Note</strong> If the top model uses a data dictionary, you cannot select this option.</td>
</tr>
<tr>
<td>Fix errors automatically</td>
<td>By default, if an advisor check finds any errors and the advisor can fix the error, the advisor provides a <strong>Fix</strong> button that you can click to have the advisor fix the issue. If you enable this parameter, the advisor fixes all conversion errors that it can, without displaying the <strong>Fix</strong> button.</td>
</tr>
<tr>
<td>Input Parameter</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------------------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Replace content of a subsystem with a Model block</td>
<td>By default, the advisor updates the original model by inserting a Model block in the model. The advisor action depends on whether you use the automatic fix options.</td>
</tr>
<tr>
<td></td>
<td>• If you use the automatic fixes, then the advisor replaces the Subsystem block with a Model block unless the automatic fixes change the input or output ports. If the ports change, then the advisor includes the contents of the subsystem in a Model block that is inserted in the Subsystem block.</td>
</tr>
<tr>
<td></td>
<td>• If you do not use the automatic fixes, then the advisor replaces the Subsystem block with a Model block.</td>
</tr>
<tr>
<td></td>
<td>Clear this parameter to have the advisor open a new Simulink Editor window that contains only a Model block that references the newly created referenced model. The advisor does not update the original model in the other Simulink Editor window.</td>
</tr>
<tr>
<td>Input Parameter</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Check simulation results after conversion</td>
<td>Compare the results of simulating the top model for the referenced model to the results of simulating the baseline model that has the subsystem.</td>
</tr>
<tr>
<td></td>
<td>To use this option, before performing the conversion, enable signal logging for the subsystem output signals of interest in the model. Set these advisor options:</td>
</tr>
<tr>
<td></td>
<td>• Model block simulation mode — Use the same simulation mode as in the original model.</td>
</tr>
<tr>
<td></td>
<td>• Replace content of a subsystem with a Model block — Enable this option.</td>
</tr>
<tr>
<td></td>
<td>• Stop time — Specify when you want the simulations to end. The default is the stop time of the top model. If the top model stop time is set to inf, the advisor stops after 10 seconds.</td>
</tr>
<tr>
<td></td>
<td>• Absolute tolerance — Specify a value if you do not want to use the default of '1e-06'.</td>
</tr>
<tr>
<td></td>
<td>• Relative tolerance — Specify a value if you do not want to use the default of '1e-03'.</td>
</tr>
<tr>
<td></td>
<td>To see the results after the conversion is complete, click View comparison results. The advisor displays the results of the comparison in the Simulation Data Inspector. For more information, see “Compare Simulation Results Before and After Conversion”.</td>
</tr>
<tr>
<td>Stop time</td>
<td>By default, the advisor uses the stop time of the top model, unless the stop time of the top model is inf. If the stop time of the top model is inf, the advisor uses a default stop time of 10. You can specify a different stop time.</td>
</tr>
<tr>
<td></td>
<td>To use this option, select Check simulation results after conversion.</td>
</tr>
<tr>
<td>Absolute tolerance</td>
<td>The absolute signal tolerance for the simulation run comparison. The default is 1e-06.</td>
</tr>
<tr>
<td></td>
<td>To use this option, select Check simulation results after conversion.</td>
</tr>
</tbody>
</table>
### Input Parameter | Description
--- | ---
Relative tolerance | The relative signal tolerance for the simulation run comparison. The default is \(1e^{-03}\). To use this option, select **Check simulation results after conversion**.
Model block simulation mode | Simulation mode for the new Model block that references the referenced model.  
- Normal (default)  
- Accelerator

After you configure the advisor, to start the conversion checks, click **Run this task**.
Performance Advisor Checks
## Simulink Performance Advisor Checks

### In this section...

- “Simulink Performance Advisor Check Overview” on page 12-3
- “Baseline” on page 12-3
- “Checks that Require Update Diagram” on page 12-3
- “Checks that Require Simulation to Run” on page 12-3
- “Check Simulation Modes Settings” on page 12-3
- “Check Compiler Optimization Settings” on page 12-4
- “Create baseline” on page 12-4
- “Identify resource-intensive diagnostic settings” on page 12-4
- “Check optimization settings” on page 12-5
- “Identify inefficient lookup table blocks” on page 12-5
- “Check MATLAB System block simulation mode” on page 12-5
- “Identify Interpreted MATLAB Function blocks” on page 12-6
- “Identify simulation target settings” on page 12-6
- “Check model reference rebuild setting” on page 12-7
- “Identify Scope blocks” on page 12-7
- “Identify active instrumentation settings on the model” on page 12-7
- “Check model reference parallel build” on page 12-8
- “Check Delay block circular buffer setting” on page 12-10
- “Check continuous and discrete rate coupling” on page 12-10
- “Check zero-crossing impact on continuous integration” on page 12-11
- “Check discrete signals driving derivative port” on page 12-11
- “Check solver type selection” on page 12-12
- “Select multi-thread co-simulation setting on or off” on page 12-13
- “Identify co-simulation signals for numerical compensation” on page 12-13
- “Select simulation mode” on page 12-14
- “Select compiler optimizations on or off” on page 12-15
- “Final Validation” on page 12-15
Simulink Performance Advisor Check Overview

Use Performance Advisor checks to improve model simulation time.

See Also

“Improve Simulation Performance Using Performance Advisor”

Baseline

Establish a measurement to compare the performance of a simulation after Performance Advisor implements improvements.

See Also

“Create a Performance Advisor Baseline Measurement”

Checks that Require Update Diagram

These checks require that Update Diagram occurs in order to run.

See Also

“Improve Simulation Performance Using Performance Advisor”

Checks that Require Simulation to Run

These checks require simulation to run in order to collect sufficient performance data. Performance Advisor reports the results after simulation completes.

See Also

“Improve Simulation Performance Using Performance Advisor”

Check Simulation Modes Settings

These checks evaluate simulation modes (Normal, Accelerator, Rapid Accelerator, Rapid Accelerator with up-to-date check off) and identify the optimal mode to achieve fastest simulation.
See Also

“What Is Acceleration?”

Check Compiler Optimization Settings

Use these checks to select compiler optimization settings for improved performance.

See Also

“Compiler optimization level”

Create baseline

Select this check to create a baseline when Performance Advisor runs. You can also create a baseline manually. A baseline is the measurement of simulation performance before you run checks in Performance Advisor. The baseline includes the time to run the simulation and the simulation results (signals logged). Before you create a baseline for a model, in the Data Import/Export pane of the Configuration Parameters dialog box:

• Select the States check box.
• Set the Format parameter to Structure with time.

See Also

“Create a Performance Advisor Baseline Measurement”

Identify resource-intensive diagnostic settings

To improve simulation speed, disable diagnostics where possible. For example, some diagnostics, such as Solver data inconsistency or Array bounds exceeded, incur runtime overheads during simulations.

See Also

• “Diagnostics”
• “Improve Simulation Performance Using Performance Advisor”
Check optimization settings

To improve simulation speed, enable optimizations where possible. For example, if some optimizations, such as Block Reduction, are disabled, enable these optimizations to improve simulation speed.

You can also trade off compile-time speed for simulation speed by setting the compiler optimization level. Compiler optimizations for accelerations are disabled by default. Enabling them accelerates simulation runs but results in longer build times. The speed and efficiency of the C compiler used for Accelerator and Rapid Accelerator modes also affects the time required in the compile step.

See Also

• “Model Configuration Parameters: Code Generation Optimization” (Simulink Coder)
• “Improve Simulation Performance Using Performance Advisor”

Identify inefficient lookup table blocks

To improve simulation speed, use properly configured lookup table blocks.

See Also

• “Lookup Tables”
• “Optimize Generated Code for Lookup Table Blocks”
• “Optimize Breakpoint Spacing in Lookup Tables”
• “Improve Simulation Performance Using Performance Advisor”

Check MATLAB System block simulation mode

In general, to improve simulation speed, choose Code generation for the Simulate using parameter of the MATLAB System block. Because data exchange between MATLAB and Simulink passes through several software layers, Interpreted execution usually slows simulations, particularly if the model needs many data exchanges.

This check identifies which MATLAB System blocks can generate code and changes the Simulate using parameter value to Code generation where possible.
While Code generation does not support all MATLAB functions, the subset of the MATLAB language that it does support is extensive. By using this Code generation, you can improve performance.

See Also

• MATLAB System
• “Simulation Modes”
• “Improve Simulation Performance Using Performance Advisor”

Identify Interpreted MATLAB Function blocks

To improve simulation speed, replace Interpreted MATLAB Function blocks with MATLAB Function blocks where possible. Because data exchange between MATLAB and Simulink passes through several software layers, Interpreted MATLAB Function blocks usually slow simulations, particularly if the model needs many data exchanges.

Additionally, because you cannot compile an Interpreted MATLAB Function, an Interpreted MATLAB Function block impedes attempts to use an acceleration mode to speed up simulations.

While MATLAB Function blocks do not support all MATLAB functions, the subset of the MATLAB language that it does support is extensive. By replacing your interpreted MATLAB code with code that uses only this embeddable MATLAB subset, you can improve performance.

See Also

• MATLAB Function
• “Improve Simulation Performance Using Performance Advisor”

Identify simulation target settings

To improve simulation speed, disable simulation target settings where possible. For example, in the Configuration Parameters dialog box, clear the Simulation Target > Echo expression without semicolons check box to improve simulation speed.
See Also

- “Model Configuration Parameters: Simulation Target”
- “Improve Simulation Performance Using Performance Advisor”

Check model reference rebuild setting

To improve simulation speed, in the Configuration Parameters dialog box, verify that the Model Referencing > Rebuild parameter is set to If any changes in known dependencies detected.

See Also

- “Rebuild”
- “Improve Simulation Performance Using Performance Advisor”

Identify Scope blocks

Opened and uncommented Scope blocks can impact simulation performance. To improve simulation performance, close and comment out Scope blocks. Right-click a scope block, and then select Comment Out.

For opened Scopes, you can improve simulation speed by reducing updates. From the Scope Simulation menu, select Reduce Updates to Improve Performance.

See Also

- “Improve Simulation Performance Using Performance Advisor”

Identify active instrumentation settings on the model

Identify active instrumentation settings on the model. The fixed-point instrumentation mode controls which objects log minimum, maximum, and overflow data during simulation. Instrumentation is required to collect simulation ranges using the Fixed-Point Tool. These ranges are used to propose data types for the model (requires Fixed-Point Designer). When you are not actively converting your model to fixed point, disable the fixed-point instrumentation to restore the maximum simulation speed to your model.

From the model Analysis menu, select Data Type Design > Fixed-Point Tool. Under System under design, click Continue.
In the Model Hierarchy pane, the Fixed-Point Tool denotes systems that currently have instrumentation turned on with (mno), or (a). Right-click the system in the model hierarchy and, under Fixed-point instrumentation mode, select Use local settings or Force off.

See Also

- Fixed-Point Instrumentation and Data Type Override (Fixed-Point Designer)

Check model reference parallel build

To improve simulation, verify the number of referenced models in the model. If there are two or more referenced models, build the model in parallel if possible.

Performance Advisor analyzes the model and estimates the build time on the current computer as if it were using several cores. It also estimates the parallel build time for the model in the same way an estimation would be performed if Parallel Computing Toolbox or MATLAB Parallel Server software were installed on the computer. Performance Advisor performs this estimate as follows:

1. Search the model for referenced models that do not refer to other referenced models.
2. Calculate the average number of blocks in each of the referenced models that do not refer to other referenced models.
3. Of the list of referenced models that do not refer to others, select a referenced model whose number of blocks is closest to the calculated average.
4. Build this model to obtain the build time.
5. Based on the number of blocks and the build time for this referenced model, estimate the build time for all other referenced models.
6. Based on these build times, estimate the parallel build time for the top model.
To calculate the overhead time introduced by the parallel build mechanism, set the Parallel Build Overhead Time Estimation Factor. Performance Advisor calculates the estimated build time with overhead as:

\[(1 + \text{Parallel Build Overhead Time Estimation Factor}) \times \text{(Build time on a single machine)}\]

**See Also**

- “Enable parallel model reference builds”
- “Improve Simulation Performance Using Performance Advisor”
**Check Delay block circular buffer setting**

To improve simulation, check that each Delay block in the model uses the appropriate buffer type. By default, the block uses an array buffer (the **Use circular buffer for state** option is not selected). However, when the delay length is large, a circular buffer can improve execution speed by keeping the number of copy operations constant.

If the Delay block is currently using an array buffer, and all of the following conditions are true, Performance Advisor selects a circular buffer:

- The Delay block is in sample-based mode, i.e., either the **Input processing** parameter is set to **Elements as channels (sample based)**, or the input signal type is set to **Sample based**.
- The value or upper limit of the delay length is 10 or greater.
- The size of the state—equal to the delay length multiplied by the total of all output signal widths—is 1000 or greater.

**See Also**

- Delay
- “Improve Simulation Performance Using Performance Advisor”

**Check continuous and discrete rate coupling**

If your model contains both discrete and continuous rates, the coupling between these rates can slow down simulation. Performance Advisor checks for these conditions in your model.

- The model is using a variable step solver.
- The model contains both continuous and discrete rates.
- The fastest discrete rate is relatively smaller than **Max step size** determined by the solver.

Setting the **DecoupledContinuousIntegration** parameter to on might speed up simulation.

**See Also**

- “Solver Selection Criteria”
• “Speed Up Simulation”
• “Improve Simulation Performance Using Performance Advisor”

Check zero-crossing impact on continuous integration

If your model contains zero-crossings which do not impact the continuous integration, the simulation might slow down when all the following conditions are satisfied:

• The model uses a variable-step solver.
• The model contains blocks that have continuous states and zero-crossings.
• Some of the zero-crossings do not affect the integration of the continuous states.

Setting the MinimalZcImpactIntegration parameter to On might speed up simulation.

See Also
• “Speed Up Simulation”
• “Solver Selection Criteria”
• “Improve Simulation Performance Using Performance Advisor”

Check discrete signals driving derivative port

Run this check if your simulation has many unnecessary resets. A discrete signal driving a block with continuous states triggers a reset at every sample time hit of the block. These resets are computationally expensive. Performance Advisor checks for these signals and blocks and provides a list of the same.

You can edit the model around the discovered discrete signals that drive these blocks to remove such cases. For example, inserting a Zero Order Hold block between the discrete signal and the corresponding block with continuous states might help resolve the issue.

See Also
• “Speed Up Simulation”
• “Modeling Techniques That Improve Performance”
Check solver type selection

To improve simulation, check that the model uses the appropriate solver type.

Explicit vs. Implicit Solvers

Selecting a solver depends on the approximation of the model stiffness at the beginning of the simulation. A stiff system has both slowly and quickly varying continuous dynamics. Implicit solvers are specifically designed for stiff problems, whereas explicit solvers are designed for non-stiff problems. Using non-stiff solvers to solve stiff systems is inefficient and can lead to incorrect results. If a non-stiff solver uses a very small step size to solve your model, check to see if you have a stiff system.

<table>
<thead>
<tr>
<th>Model</th>
<th>Recommended Solver</th>
</tr>
</thead>
<tbody>
<tr>
<td>Represents a stiff system</td>
<td>ode15s</td>
</tr>
<tr>
<td>Does not represent a stiff system</td>
<td>ode45</td>
</tr>
</tbody>
</table>

Performance Advisor uses the heuristic shown in the table to choose between explicit and implicit solvers.

<table>
<thead>
<tr>
<th>Original Solver</th>
<th>Performance Advisor Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variable step solver</td>
<td>Calculates the system stiffness at 0 first. Then:</td>
</tr>
<tr>
<td></td>
<td>• If the stiffness is greater than 1000, Performance Advisor chooses ode15s.</td>
</tr>
<tr>
<td></td>
<td>• If the stiffness is less than 1000, Performance Advisor chooses ode45.</td>
</tr>
<tr>
<td>Fixed-step continuous solver</td>
<td>• If the stiffness is greater than 1000, Performance Advisor chooses ode14x.</td>
</tr>
<tr>
<td></td>
<td>• If the stiffness is less than 1000, Performance Advisor chooses ode3.</td>
</tr>
</tbody>
</table>

This heuristic works best if the system stiffness does not vary during simulation. If the system stiffness varies with time, choose the most appropriate solver for that system rather than the one Performance Advisor suggests.
See Also

- “Solver Selection Criteria”
- “Speed Up Simulation”
- “Improve Simulation Performance Using Performance Advisor”

Select multi-thread co-simulation setting on or off

Adjust co-simulation settings for better performance and accuracy.

- Validate and revert changes if simulation time increases — Performance Advisor reverts previous co-simulation settings when the simulation time increases.
- Validate and revert changes if degree of accuracy is greater than tolerance — Performance Advisor reverts previous co-simulation settings if the degree of accuracy is greater than tolerance.

Tip You can use the tic and toc functions to measure the simulation time.

See Also

- tic
- toc
- sim

Identify co-simulation signals for numerical compensation

Identify co-simulation signals that may need explicit numerical compensation.

- Validate and revert changes if time of simulation increases — Performance Advisor reverts previous co-simulation settings the simulation time increases.
- Validate and revert changes if degree of accuracy is greater than tolerance — Performance Advisor reverts co-simulations if the degree of accuracy is greater than tolerance.

Tip You can use the tic and toc functions to measure the simulation time.
Select simulation mode

To achieve fastest simulation time, use this check to evaluate the following modes and identify the optimal selection:

- Normal
- Accelerator
- Rapid Accelerator
- Rapid Accelerator with up-to-date check off

In Normal mode, Simulink interprets your model during each simulation run. If you change the model frequently, this is generally the preferred mode to use because it requires no separate compilation step. It also offers the most flexibility to make changes to your model.

In Accelerator mode, Simulink compiles a model into a binary shared library or DLL where possible, eliminating the block-to-block overhead of an interpreted simulation in Normal mode. Accelerator mode supports the debugger and profiler, but not runtime diagnostics.

In Rapid Accelerator mode, simulation speeds are fastest but this mode only works with models where C-code is available for all blocks in the model. Also, this mode does not support the debugger or profiler.

When choosing Rapid Accelerator with up-to-date check off, Performance Advisor does not perform an up-to-date check during simulation. You can run the Rapid Accelerator executable repeatedly while tuning parameters without incurring the overhead of up-to-date checks. For instance, if you have a large model or a model that makes extensive use of model reference, this method of execution can increase efficiency.

For models with 3-D signals, Normal or Accelerator modes work best.
See Also

- “How Acceleration Modes Work”
- “Choosing a Simulation Mode”
- “Comparing Performance”
- “Run Simulations Programmatically”

Select compiler optimizations on or off

Use this check to determine whether performing compiler optimization can help improve simulation speed. The optimization can only be performed in Accelerator or Rapid Accelerator modes.

**Note** This check will be skipped if MATLAB is not configured to use an optimizing compiler.

See Also

- “How Acceleration Modes Work”
- “Choosing a Simulation Mode”
- “Comparing Performance”
- “Improve Simulation Performance Using Performance Advisor”

Final Validation

This check validates the overall performance improvement of simulation time and accuracy in a model. If the performance is worse than the original model, Performance Advisor discards all changes to the model and loads the original model.

Global settings for validation do not apply to this check. If you have not validated the performance improvement from changes resulting from other checks, use this check to perform a final validation of all changes to a model.

See Also

- “Comparing Performance”
• “Improve Simulation Performance Using Performance Advisor”
Simulink Limits

- “Maximum Size Limits of Simulink Models” on page 13-2
- “Systems and Models” on page 13-3
- “Simulink Models” on page 13-7
- “Simulink Simulation” on page 13-35
- “Simulink Tools” on page 13-41
- “Programming Constructs in Simulink” on page 13-55
- “Model Development Processes” on page 13-58
## Maximum Size Limits of Simulink Models

The following table documents some limits on the size and complexity of Simulink models.

<table>
<thead>
<tr>
<th>Model Feature</th>
<th>Limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum number of levels in a block diagram</td>
<td>1024</td>
</tr>
<tr>
<td>Maximum number of branches in a line</td>
<td>1024</td>
</tr>
<tr>
<td>Maximum length of a parameter name</td>
<td>63</td>
</tr>
<tr>
<td>Maximum length of a parameter character vector value</td>
<td>32768</td>
</tr>
<tr>
<td>Maximum size of a model window, as determined by supported coordinates</td>
<td>-1073740824 to 1073740823, inclusive</td>
</tr>
<tr>
<td>Maximum number of bytes for a single logged data point</td>
<td>90 MB</td>
</tr>
<tr>
<td>Maximum number of bytes of logged simulation data</td>
<td>Limited by available hard drive space</td>
</tr>
<tr>
<td>Maximum number of bytes for an individual block I/O buffer length</td>
<td>$2^{31-1}$ bytes on 64-bit systems</td>
</tr>
<tr>
<td>Maximum number of bytes for the total block I/O buffer length in a model</td>
<td>$2^{64-1}$ bytes on 64-bit systems</td>
</tr>
<tr>
<td>Maximum length of integer and fixed-point data types</td>
<td>128 bits</td>
</tr>
<tr>
<td>Maximum length of string data type</td>
<td>32,766 characters</td>
</tr>
</tbody>
</table>
Systems and Models

The concepts in this topic provide a consistent and common language for using Simulink software tools.

**System**

A system is a group of interdependent physical and functional parts with measurable characteristics that change over time.

For example, a vehicle is a system with multiple parts. Measurable characteristics include the linear speed of the vehicle and the rotational speed of the wheels.

**System Component**

A system component is part of a system that interacts with the other parts of the system. The interactions between components define the structure and behavior of the system.

For example, a cruise control module is a system component in a vehicle system. A microcontroller and the hardware associated with it define the structure while a software algorithm to control speed defines the behavior.

**Model**

A model is a mathematical description of a system derived either from physical laws or experimental data. The description typically uses a set of variables with a set of differential and difference equations that define the relationships between the variables.

In the following example for a vehicle, \( u(t) \) is the force (N) moving a vehicle forward, \( v(t) \) is the velocity (m/s), \( b \) is a drag coefficient (N·s/m), and \( m \) is the mass of the vehicle (kg).
The vehicle is a continuous system. For continuous systems, differential equations describe the rate of change for variables with the equations defined for all values of time. The velocity of the vehicle \( v(t) \) and its acceleration \( v'(t) \) are defined with the following first order differential equation.

\[
mv'(t) + bv(t) = u(t)
\]

You can create a Simulink model for this equation by adding blocks, specifying block behavior, and using signal lines to connect the blocks to each other. The following Simulink block diagram implements the differential equation.

![Simulink block diagram](image)

**Model Component**

A model component is part of a model that interacts with the other parts through an interface of inputs and outputs. Simulink implements model components using Subsystem and Model blocks. A Model block references another Simulink model saved in a separate file.

In the following example, the control model was saved in the Simulink model file `control_model.slx`, and then referenced from a Model block in a second Simulink model. A Subsystem block was added for modeling the vehicle mechanics.
Typically, controllers are built with discrete systems using a computer to implement the control algorithm. For discrete systems, difference equations describe the rate of change for variables defined only at specific times. For example, the control signal for a simple discrete PI (proportional–integral) controller can be defined with the following difference equation.

\[ \text{PI}[n] = e[n]Kp + (e[n]+\text{integral}[n-1])Ki \]

Where \( e[n] \) is the error between a signal whose value is controlled (velocity) and the specified value (set velocity), \( Kp \) is the proportion constant, \( Ki \) is the integration constant, and \( n \) is the time step.

The following Simulink block diagram implements the difference equation.

![Simulink block diagram](image)

See also: “Model a Continuous System”, “Component-Based Modeling Guidelines”, “Create a Custom Library”, “Model Reference Basics”.

**Differential Algebraic Equations**

Some systems of equations contain additional constraints that involve the independent variable and the state vector in addition to differential equations. Such systems are called differential algebraic equations (DAEs).

The term algebraic refers to equations that do not involve any derivatives.

In Simulink models, algebraic loops are algebraic constraints. Models with algebraic loops define a system of differential algebraic equations.

For example,

\[ x' = x \]
\[ 0 = -x + u - 2x \]

is a system of differential algebraic equations implemented in the following Simulink model.
Simulink Models

A model is an abstract and simplified description of a system using mathematical equations and diagrams. The modeling concepts in this topic provide context for understanding the process of mathematically describing a system with Simulink software tools.

Block Diagram

A block diagram is a visual representation of a model in the Simulink Editor. The editor allows you to add blocks selected from block libraries representing elementary model components. Elementary components include integrator, gain, and sum blocks. Blocks are connected to each other with signal and event lines to visually construct the model equations.

Block Diagram Semantics

A classic block diagram of a system is drawn graphically with blocks and lines. The history of these block diagrams is derived from engineering areas such as Feedback Control Theory and Signal Processing. A block within a block diagram defines a model in itself. The relationships between elementary models are represented by signal lines connecting the blocks. Collectively the blocks and lines in a block diagram describe the overall model of a system.
Simulink extends classic block diagrams by:

- Adding a set of equations (block methods) to each block that define the time-based relationships between the input signals, output signals, and the block state variables.
- Adding parameters to each block that specify the coefficients from the model equations.
- Providing a Simulink engine for numerical solution of the block diagram by evaluating the relationships over time, where time starts at a user specified “start time” and ends at a user specified “stop time.”

See also: “Simulink Block Diagrams”, “Build and Edit a Model in the Simulink Editor”, Use block diagrams to graphically represent dynamic systems, “Simulation”

**Blocks**

A block is a basic modeling construct of the Simulink Editor. You add instances of the blocks from the build-in Simulink libraries to perform specific operations. You can also create your own custom blocks. Some blocks have input signals, output signals, and state. Most blocks have parameters that you use to specify block behavior. Whether a block has parameters and the nature of those parameters is specific to each block.

Each block represents a set of equations for the Simulink engine. The equations are represented as block methods. The blocks shown above have the following block methods.
Block methods are evaluated during the simulation of a block diagram. The evaluation of the block methods is performed within a simulation loop, where each cycle through the simulation loop represents the evaluation of the block diagram at a given point in time.

**Virtual Blocks**

Virtual blocks organize and provide graphical hierarchy in a model, but they do not affect the simulation result. During model simulation, the Simulink engine flattens all virtual blocks. That is, Simulink expands the blocks in place before execution. This expansion is similar to the way macros work in a programming language such as C or C++.

Simulink defines the following virtual blocks:

- **Virtual Subsystem** – Use a virtual subsystem to encapsulate related and functional parts within a larger model. A Virtual Subsystem block has the check box for the parameter **Treat as atomic unit** cleared.

- **Inport and Outport** – Use port blocks to move data (signals) and events (function-calls) from outside a System block or referenced Model block to within the block.
• Bus Creator, Bus Assignment, Bus Selector - Bus blocks combine signals into a virtual bus (composit signal) and manage the routing of signals around a complex block diagram. Virtual bus signals are simply a graphical convenience and do not change the behavior of the model.

See “Nonvirtual and Virtual Blocks”, “Virtual Signals”, “Types of Composite Signals”.

**Non-Virtual Blocks**

Non-virtual blocks provide execution control and graphical hierarchy in a model. Simulink defines the following non-virtual Subsystem and Model blocks:

• Atomic Subsystem and Model - Blocks within an Atomic Subsystem block or a Model block that references a model execute as a single unit (atomic execution) at each time step. For an Atomic Subsystem block, the check box for the parameter **Treat as atomic unit** is selected.
You can place any Simulink block in an Atomic Subsystem or referenced model, including blocks with different execution rates. This flexibility provides the advantage of grouping functional aspects of a model at the execution level.

In the following example, the vehicle model is an Atomic Subsystem block containing blocks that model the mechanics of a car. The control model is referenced from a Model block.

- **Enabled and Triggered** – Atomic Subsystem or Model block whose execution is controlled external data from a signal. Contains an Enable or Trigger port block. For the Trigger port block, the parameter **Trigger type** is set to *rising*, *falling*, or *either*.

- **Function-Call** – Atomic Subsystem or Model block whose execution is controlled by an event from a function-call initiator (e.g., Stateflow chart, Function-Call Generator, and S-Function blocks). Contains a Trigger port block with **Trigger type** set to *function-call*. 
Simulink defines the following non-virtual components for only Subsystem blocks.

- **Action Subsystem** – Atomic Subsystem block whose execution is controlled by an event from an action initiator (e.g., If or Switch Case block). Contains an Action Port block within the Subsystem block.

- **While Iterator Subsystem** – Atomic Subsystem block that runs multiple iterations during each model time step. Contains a While Iterator block that controls the number of iterations by evaluating a logical condition.

A while iterator subsystem is similar to a function-call subsystem in that it can run for any number of iterations at a given time step. The while iterator subsystem differs from a function-call subsystem in that there is no separate initiator.
• For Iterator Subsystem – Atomic Subsystem block that runs a fixed number of iterations during each model time step. Contains a For Iterator block that controls the number of iterations.

See “Nonvirtual and Virtual Blocks”.

**Block Mask**

A block mask is a custom block parameter interface that displays only selected block parameters. A mask for a Subsystem block allows you to provide an interface for setting parameters on blocks inside the Subsystem block without having to navigate the model hierarchy.

See “Masking Fundamentals”.

**Custom Blocks**

Custom blocks are new blocks that extend the built-in model functionally of Simulink. You can create libraries of custom blocks to use in your model.

The following blocks define the algorithm of a custom block either graphically with a block diagram or programmatically.

• MATLAB Function – Code a MATLAB function using the MATLAB language that runs in a Simulink model. See “Implementing MATLAB Functions Using Blocks”.

• MATLAB System – Bring existing System objects based on matlab.System into Simulink®. See “MATLAB System Block”.

• Subsystem – Draw a block diagram representing an algorithm, wrap this diagram in an instance of the Simulink Subsystem block, and then provide the block with a parameter dialog using a Simulink block mask. See Subsystem, Atomic Subsystem, CodeReuse Subsystem.

• C Caller – Integrates your external C code into a Simulink® model. See “Integrate C Code Using C Caller Blocks”.

• S-function – Create a custom block programmatically by creating a MATLAB file or a MEX-file that contains the block’s system functions. The resulting file is called an S-function. You then associate the S-function with instances of the Simulink S-Function block in your model. See S-Function.

See “Types of Custom Blocks”, “Block Authoring and Simulation Integration”.

Lines

A line is a basic modeling construct of the Simulink Editor. Lines connect the output ports of blocks with the input ports of other blocks.

Signal Lines

Signal lines transfer data during a simulation from one block to another block. Signals are time varying quantities that have values at all points in time (continuous) or at specified time points (discrete). The source of a signal corresponds to the block that writes to the signal during evaluation of its block output method. The destination of a signal is the block that reads the signal during the evaluation of its block input method.

You can specify signal attributes, including signal name, data type (e.g., double, 32-bit integer), numeric type (e.g., real, complex) and dimension (e.g., one-dimensional, two-dimensional, multi-dimension array). Many blocks can accept or output a signal of any data type or dimensionality. Other blocks impose restriction on the attributes of the signal they can handle.

During a simulation, save data from a signal by logging the signals.

Event Lines

In response to a function-call or action initiator, event lines send an event to a Subsystem or Model block. The block that receives the event, executes the blocks within it one or more times during a time step. Function-call initiators include Function-Call Generator blocks, Stateflow charts, and S-function blocks. Action initiators include If and Switch Case blocks.
See “Signals” and “Signal Basics”.

Data

Data is the parameter and input signal values that Simulink uses to produce the output values resulting from simulating the behavior of a model.

Types of data:

- Model parameters - Variables to specify block and signal parameters in a model. Includes MATLAB variables, parameter and signal data objects, data type objects, and bus objects.
- Model configuration parameters - Control a simulation.
- Simulation data - Input data that you use to drive a simulation and the output data that a simulation generates.
- Data Objects - Use the Simulink.Signal and Simulink.Parameters classes to create data objects. See “Data Objects”.

Location of data:

- **Block parameters** - Use block parameters to directly specify numerical values. You can also enter a variable name and define its value in the Simulink model workspace, a Simulink data dictionary, or the MATLAB base workspace.
MATLAB base workspace - Use the MATLAB base workspace to save variables that are separate from the model in a MAT-file or MATLAB script.

Simulink treats the base workspace and the data dictionary as a single global name space. If the base workspace and a referenced data dictionary have identical variable name, Simulink uses the variable value in the data dictionary.

The MATLAB base workspace contains variables that are global and visible to all Simulink models.
See “Comparison of Signal Loading Techniques”, “Export Simulation Data”.

- **Simulink model workspace** – Use the model workspace to define and store local data variables. Variables defined in a model workspace are visible only within the scope of the model with a unique name space. Therefore, you can use the same variable name in multiple model workspaces and you can assign a unique variable value to the name in each model.

Define variables in the data dictionary using Model Explorer. In the **Modeling** tab, click **Model Explorer**. In the left pane, select Model Workspace.
Values for the model workspace variables are initialized from values saved with the model, from a separate MAT-file or MATA LB file, or initialized using MATLAB code saved with the model.

See “Model Workspaces”, “Specify Source for Data in Model Workspace”.

- **Simulink data dictionary** - Use a Data Dictionary to define and store global data, share data between models, and track changes made to the data. Data is saved in a file separate from the model.

Create a data dictionary file. In the **Modeling** tab, under **Design**, click **Data Dictionary**. Click **New** and enter a file name with an extension `.sldd`. 
Define variables in the data dictionary using Model Explorer. In the **Modeling** tab, click **Model Explorer**. In the left pane, select **Design Data**.

See “What Is a Data Dictionary?”.  

- Imported and exported data – Import signals for a simulation from the MATLAB base workspace, a MAT-file, or a spreadsheet. Create input signals using a source block or the Signal Editor block. Use signal logging to export simulation results. Signals are saved as vectors \([t, X, Y]\) for time, state and output at each major time step.
Parameters

Parameters are Simulink model characteristics that affect the simulation and code generation result.

Model Parameters

Model parameters specify model behavior during compilation, simulation, and code generation such as the solver to use and the types of errors and warnings to display.

To specify model parameters, in the Modeling tab, click Model Settings. The Configuration Parameters dialog box opens.

See also: “Determine Where to Store Variables and Objects for Simulink Models”, “Symbol Resolution”.

See “Configuration Parameters Dialog Box Overview”, .
**Block Parameters**

Block parameters define model dynamics and mathematics. Whether a block has parameters that you can set and the nature of those parameters is specific to each block. To specify block parameters, open the block parameter dialog box, or in the **Modeling** tab, under **Design**, click one of the following:

- **Property Inspector** – Specify block parameters directly or enter a variable name for a parameter value.
- **Model Data Editor** – Specify block parameters with variables.
- **Model Explorer** – Specify block parameters with variables.

See “Set Block Parameter Values”.

**Block Parameter Variables**

If you have a single parameter used in many locations across a large model, updating every instance of that parameter can be a difficult task. Instead, enter a variable name as the value of a parameter and then define the variable once using one of the following methods.

- **Simulink Model Workspace** – Create model workspace variables and assign values to the variables. The parameters you define are specific to the model and saved with the model. Use a MATLAB variable for ease of maintenance. Use a Simulink.Parameter object to control additional properties including, data type, dimensions, and units. See “Edit and Manage Workspace Variables by Using Model Explorer”.
- **Simulink Data Dictionary** – Create design data variables in a data dictionary and link the dictionary to a model. During model simulation, Simulink retrieves data from the data dictionary.
- **MATLAB Base Workspace** – In the MATLAB base workspace, define parameters using any MATLAB mechanism for defining a variable. For example, you can use a MAT-file and load the variables when you open the model. Use MATLAB expressions to specify parameter variable values. Simulink evaluates the expressions before running a simulation. See “Save and Load Workspace Variables” (MATLAB), “Save Workspace Variables” (MATLAB), “Load Workspace Variables” (MATLAB).

Using the MATLAB base workspace or a Simulink data dictionary to define variables is useful when you are using the same set of parameters for more than one model. This mechanism also allows you to use different sets of parameter values for the same model.
Model Argument and Instance Parameters

When you reference the same model from multiple Model blocks you create instances of the model. You can set block parameters to use the same value or a different value for each instance of the model.

To use different values,

1. In the model workspace for the referenced model, create a MATLAB variable or Simulink.Parameter object.
2. Enter a parameter name and a default parameter Value. Select the Argument check box to create a model argument.
3. For a block in the referenced model, enter the name of the model argument for the value of a block parameter.
4. For each Model block, open the block parameters dialog box, select the Instance parameters tab, enter a value for the instance parameter with the model argument name.

See also “Parameterize Instances of a Reusable Referenced Model”. “Tunable Block Parameters” on page 13-22

Tunable Block Parameters

You can change the value of a tunable block parameter during a simulation. This allows you to determine interactively the most suitable value for a parameter. When you change the value of a tunable parameter, the change takes effect at the start of the next time step. For example, the gain parameter of the Gain block is tunable. You can change the block gain while a simulation is running. See “Tune and Experiment with Block Parameter Values”.

Setting Block Parameters with the Property Inspector

To specify parameters, in the Modeling tab, under Design, click Property Inspector. Click a block to display the block parameters and properties.
See also “Specify Block Properties”.

**Property**

Properties are Simulink model characteristics that generally do not affect the simulation result. To specify properties, in the **Modeling** tab, under **Design**, click **Property Inspector**.

**Model Properties**

With the Property Inspector opened, click a blank space within a block diagram, or in the **Modeling** tab, select **Model Settings > Model Properties**. Model properties include:

- General – Name an location of model file.
- Design Data – Variables defined outside the model that parameterize the model and its blocks and signals.
- Callbacks – Commands that execute when a specific model event occurs.
See “Manage Model Properties”.

**Block Properties**

With the Property Inspector opened, select a block, and then select the **Properties** tab. Block properties include:

- **Block Annotation** – Values of selected block parameters displayed below the block.
- **Callbacks** – Commands that execute when a specific block event occurs. For example, you can set up a MATLAB script to load and define block parameter variables using expressions.
- **Priority** – Set the relative execution order of blocks. A lower value executes the block first before a larger priority value.
- **Tags** – Block identifiers that are programmatically searchable.

**Setting Model and Block Properties with Property Inspector**

To set properties, in the **Simulation** tab, under **Prepare**, click **Property Inspector**. Click an empty space to display the model properties. Click a block to display the block properties.
State variable

The state of a model is defined by the values of its state variables. State variables are the set of variables whose values at time zero with the values of model inputs and the model equations can determine the behavior of a model during a simulation. Examples of state variables include, motor position and velocity, inductor current, capacitor voltage, solution temperature, and gas pressure.

If the current output value of a block is a function of the previous output value, the block defines a state variable that needs to be saved between time steps. Computing a block output hence entails saving the value of the state variable at the current time step for use in computing the output at a subsequent time step.

Working with States

The following facilities are provided for determining, initializing, and logging model states during simulation:

See also: “Specify Block Properties”, “Callbacks for Customized Model Behavior”.
The **Data Import/Export** pane of a model's Configuration Parameters dialog box (see “State Information”) allows you to specify initial values for model states, and to record the values of the states at each time step during simulation as an array or structure variable in the MATLAB workspace.

Open the Configuration Parameters dialog box. Select the **States** check block for logging to the MATLAB variable `xout`.

![Configuration Parameters dialog box](image)

In the MATLAB Command Window, list the logged values for state of the Integrator block at each time step.

```
>> xout{1}.Values.Data
```

```
ans =
    0.0000
    1.8127
    3.2968
    4.5119 . . .
```

The **model** command displays information about the states defined by a model, including the total number of states, the block that defines each state with its initial value. List blocks in the model `vehicle_model` with states.
In the MATLAB Command Window, enter the model function, and then list blocks with state.

```matlab
[sys,x0,str,ts] = vehicle_model([],[],[],'sizes')
str =
    {'vehicle_model/Integrator'}
```

- The Simulink debugger displays the value of a state at each time step during a simulation, and the Simulink debugger's `states` command displays information about the model's current states (see “Simulink Debugger”).

In the MATLAB Command Window, start the Simulink debugger, and then enter the command `states`.

```matlab
sldebug 'vehicle_model'
(sldebug @0): >> states
```

**Continuous States for 'vehicle_model':**

<table>
<thead>
<tr>
<th>Idx</th>
<th>Value</th>
<th>(system:block:element Name 'BlockName')</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>(0:0:0 CSTATE 'vehicle_model/Integrator')</td>
</tr>
</tbody>
</table>

- The Block Parameters dialog box (and the `ContinuousStateAttributes` parameter) allows you to give names to states for those blocks (such as the Integrator) that employ continuous states. This can simplify analyzing data logged for states, especially when a block has multiple states.
**Continuous States**

A continuous state is defined for all values of time. An example of a continuous state would be the velocity of a car show on an analog speedometer with a needle position that changes continuously with the rotation of a tire.

Blocks with continuous states and the block parameter **Initial condition** include:

- Integrator
- State-Space
- Transfer Fcn

In general, excluding simple models, analytical methods do not exist for integrating states represented by ordinary differential equations. Integrating the states requires the use of numerical methods.

**Discrete States**

A discrete state is defined only at specific times. It is an approximation of a continuous state where the state is updated at periodic or aperiodic time intervals. An example of a discrete state would be the velocity of a car shown on a digital speedometer that is updated every second as opposed to continuously.

Blocks with discrete states include:

- Discrete Time Integrator
- Discrete State-Space
- Discrete Transfer Fcn
- Delay

Computing the discrete state for a block requires knowing its value at the previous time step and also the current input value to the block. Simulink provides two types of discrete solvers:

- Fixed-step discrete solver – determines a fixed step size that hits all the sample times for all the model’s discrete states, regardless of whether the states actually change value at the sample time hits.
- Variable-step discrete solver – varies the step size to ensure that sample time hits occur only at times when the states change value.
**Modeling Hybrid Systems**

A hybrid model has both continuous and discrete states. Solving such a model entails choosing a step size that satisfies both the precision constraint on the continuous state integration and the sample time interval constraint on the discrete states. Simulink meets this requirement by passing the next sample time interval for the discrete solver as an additional constraint on the continuous solver. The continuous solver chooses a step size that advances the simulation up to but not beyond the time of the next discrete sample time. The continuous solver can take a time step short of the next sample time hit to meet its accuracy constraint but it cannot take a step beyond the next sample time hit even if its accuracy constraint allows it to.

You can simulate hybrid systems using any of the integration methods, but certain methods are more effective than others. For most hybrid systems, ode23 and ode45 are superior to the other solvers in terms of efficiency. Because of discontinuities associated with the sample and hold of the discrete blocks, do not use the ode15s and ode113 solvers for hybrid systems.

**Sample Time**

Sample time is the time interval that specifies the rate (1 / sample time) for executing block methods to produce outputs and update internal state of blocks. Time is an inherent component of a block diagram in that the simulation results of a block diagram change with time.

Sample time is specified as:

- Continuous - Blocks run at variable times based on solver settings.
- Discrete - Blocks run at specific times that are explicitly specified.

In the following example, the controller runs at a discrete rate specified with a sample time of 0.01 seconds while Simulink determined the vehicle model to have a continuous sample time.
Determining a system's behavior over time thus entails repeatedly solving the model at intervals, called time steps or time intervals, from the start of the time span to the end of the time span. The process of solving a model at successive time steps is referred to as simulating the system that the model represents.


**Block Sample Times**

Every Simulink block has a sample time which defines when the block will execute. Most blocks allow you to specify the sample time via a SampleTime parameter. Common choices include discrete, continuous, and inherited sample times.
For discrete blocks, the sample time is a vector \([T_s, T_0]\) where \(T_s\) is the time interval or period between consecutive sample times and \(T_0\) is an initial offset to the sample time. In contrast, the sample times for nondiscrete blocks are represented by ordered pairs that use zero, a negative integer, or infinity to represent a specific type of sample time. For example, continuous blocks have a nominal sample time of \([0, 0]\) and are used to model systems in which the states change continuously (e.g., a car accelerating). Whereas you indicate the sample time type of an inherited block symbolically as \([-1, 0]\) and Simulink then determines the actual value based upon the context of the inherited block within the model.

Note that not all blocks accept all types of sample times. For example, a discrete block cannot accept a continuous sample time.

For a visual aid, Simulink allows the optional color-coding and annotation of any block diagram to indicate the type and speed of the block sample times. You can capture all of the colors and the annotations within a legend (see “View Sample Time Information”).

For a more detailed discussion of sample times, see “Sample Time”

**Units**

Units are used to measure the total amount of a quantity.

Simulink units are specified as a Inport block or Outport block parameter at the boundaries of a Simulink model component. Simulink model components include Subsystem blocks, Model blocks, Stateflow charts, and Simulink to Simscape converter blocks. To display units on a model, in the **Debug** tab, select **Information Overlays > Port Units**.
See also: “Unit Specification in Simulink Models”, “Converting Units”.

**Direct Feedthrough**

Simulink block characteristic where the output port signal of a block is computed from the values of its input port signals. The output signal value is a function of the input signal values.

Blocks with direct feedthrough include the Gain, Product, Sum, Transfer Fcn, State-Space, and Math Function blocks.

See also: “Algebraic Loop Concepts”.

**Algebraic Loop**

*Signal* loop between blocks with *direct feedthrough*. An algebraic loop generally occurs when an input port of a block with *direct feedthrough* is driven directly by the output port of the same block or indirectly through other blocks with *direct feedthrough*.

In the following model, the two Gain blocks with *direct feedthrough* create an algebraic loop.
Artificial Algebraic Loops

An artificial algebraic loop occurs when an atomic subsystem or Model block causes Simulink to detect an algebraic loop, even though the contents of the subsystem do not contain a direct feedthrough from the input to the output. When you create an atomic subsystem, all Inport blocks are direct feedthrough, resulting in an algebraic loop.

Start with the included model, which represents a simple proportional control of the plant described by

\[
G(s) = \frac{1}{s^2 + 2s + 1}
\]

which can be rewritten in state-space form as

\[
\dot{x} = \begin{bmatrix} -2 & -1 \\ 1 & 0 \end{bmatrix} x + \begin{bmatrix} 0 \\ 1 \end{bmatrix}
\]

\[
y = \begin{bmatrix} 0 & 1 \end{bmatrix} x
\]

The system has neither algebraic variables nor direct feedthrough and does not contain an algebraic loop.
Modify the model as described in the following steps:

1. Enclose the Controller and Plant blocks in a subsystem.
2. In the subsystem dialog box, select Treat as atomic unit to make the subsystem atomic.
3. In the Diagnostics pane of the Model Configuration Parameters, set the Algebraic Loop parameter to error.

When simulating this model, an algebraic loop occurs because the subsystem is direct feedthrough, even though the path within the atomic subsystem is not direct feedthrough. Simulation stops with an algebraic loop error.

**Zero-Crossing Detection**

The simulation uses a technique known as zero-crossing detection to accurately locate a discontinuity without resorting to excessively small time steps during a simulink. Usually this technique improves simulation run times.
Simulink Simulation

Simulating the model of a dynamic system allows you to gain insight about the behavior of a proposed system design without the time consuming process of actually building the system. The concepts in this topic provide a context for understanding how to control a model simulation with Simulink software tools.

Compilation

Compilation is the Simulink process where the block diagram is translated to an internal representation that interacts with the Simulink engine.

There are no model-level sets of differential equations that are solved numerically as a whole. Instead, the model-level equations correspond to the individual block equations that are solved numerically in a specific order.

Block Methods

The functionality of a single block is defined by multiple equations. These equations are represented as block methods. These block methods are evaluated (executed) during the execution of a block diagram. The evaluation of these block methods is performed within a simulation loop, where each cycle through the simulation loop represent the evaluation of the block diagram at a given point in time. Common block methods include:

- **Derivative** – Computes the derivatives of the block's continuous states at the current time step, given the block inputs and the values of the states at the previous time step.
- **Update** – Computes the value of the block's discrete states at the current time step, given its inputs at the current time step and its discrete states at the previous time step.
- **Output** – Computes the outputs of a block given its inputs at the current time step and its states at the previous time step.

\[
\begin{align*}
X &= [x_c, x_d] \\
\dot{x}_c &= f_c(X, U, t) \\
x_d[n + 1] &= f_d(X, U, t) \\
Y &= [y_1, y_2] \\
Y &= g(X, U, t) \\
X(t_0) &= X_0
\end{align*}
\]

- Model states
- Derivative method
- Update method
- Output method
- Initial model states
**Model Method**

In addition to block methods, a set of methods is provided that compute the model properties and its outputs. The Simulink software similarly invokes these methods during simulation to determine a model's properties and its outputs. The model methods generally perform their tasks by invoking block methods of the same type. For example, the model Outputs method invokes the Outputs methods of the blocks that it contains in the order specified by the model to compute its outputs. The model Derivatives method similarly invokes the Derivatives methods of the blocks that it contains to determine the derivatives of its states.

See also: “Simulation Phases in Dynamic Systems”.

**Callback**

Callbacks are MATLAB expressions that execute in response to a specific modeling action. Simulink provides model, block, and port callback parameters that identify specific kinds of modeling actions. You provide the code for callback parameters. Simulink executes the callback code when the associated modeling action occurs.

**Model Callback**

Model callback parameters include:

- **PreloadFcn** – Executes before a model loads. For example, you can provide code that loads the variable values a model uses into the MATLAB workspace.

See “Model Callbacks”.

**Block Callback**

Block callback parameters include:

- **OpenFcn** – Execute when you open a Subsystem block.
- **LoadFcn** – Execute after a diagram is loaded. For a Subsystem blocks, also execute block callback parameters for the blocks within Subsystem block.

**Port Callback**

Port callback parameter:
• **ConnectionCallback** - Execute code every time the connectivity of a port changes.

  See “Port Callbacks”.

### Execution Order

The Execution order is the sequence in which block output methods are called after evaluating direct feedthrough of each input port. To display execution order, in the **Debug** tab, select **Information Overlays > Execution Order**.

In the following model, the Integrator block output runs first, and then the loop of blocks connected to the Integrator block input. Missing execution numbers in a sequence are usually due to hidden buffer blocks.

See also: “Control and Display the Execution Order”, “Simulation Phases in Dynamic Systems”.

### Simulation

Simulation is the process after model **compilation** where block method outputs and states are computed at successive time steps over a specified time range using a numerical solver.

During each simulation loop, Simulink calculates a $\Delta t$ to determine the time step $t(k+1) = t(k) + \Delta t$. The size of $\Delta t$ is based on an estimated error between the simulated solution and the actual solution. At the end of a simulation, data results are given as vectors $[t, X, Y]$ for time, state and output at each time step.
See also: “Simulation Phases in Dynamic Systems”, “Simulate a Model Interactively”, “Speed Up Simulation”.

**Solver**

A Solver finds an approximate solution for a set of model equations. Simulink uses established numerical solvers for this task.

Solver step size can be fixed or variable:

- **Fixed step** - Time step $T(k+1) = T(k) + \Delta t$ where $\Delta t$ is constant. If step size is too large, simulation results can have a large error. In the following example, a step size of 2 distorts the shape of a sine wave signal. You can specify the size of the time step in the case of fixed-step solvers, or the solver can automatically determine the step size in the case of variable-step solvers.
Variable step - Variable step solvers iterate to reach a solution based on an error tolerance. Time step \( T(k+1) = T(k) + \Delta t_k \) where \( \Delta t_k \) changes from one simulation step to the next depending on the estimated error. Smaller time steps increase the accuracy of the simulation results. To minimize the computation workload, a variable-step solver chooses the largest step size consistent with achieving an overall level of precision specified by the error tolerance and observing zero-crossings. This ensures that all model states are computed to the accuracy specified by the user.

Choosing a solver method depends on the nature of the model equations. Euler's method is a simple numerical solver that calculates the next value of \( y \) by using the slope \( (y') \) of a tangent line to \( y \). If \( y \) is a function that integrates a ramp function \( x \) with a slope of 1, \( y' = x \), and a numerical solver would use the following equations.

\[
\begin{align*}
x[n+1] &= x[n] + \Delta t \times 1 \\
y[n+1] &= y[n] + \Delta t \times x[n+1]
\end{align*}
\]

Decreasing the step size increases the accuracy of the results, but it increases the time to complete a simulation. In the following example, a step size of 2 shows an error of about 20 percent after 10 seconds while a step size of 0.5 produces a result that is closer to the actual solution.
See also: “Simulation Phases in Dynamic Systems”, “Compare Solvers”, “Check and Improve Simulation Accuracy”, “Choose a Solver”. 
Simulink Tools

**Code Generation Advisor** - Simulink tool that checks a model for how objectives such as efficiency, traceability, and safety, map to code generation options in a model configuration set. To open and run the advisor, in the **Modeling** tab, click **Model Advisor**, and then click **Code Generation Advisor**.

![Code Generation Advisor](image)

**Configuration Parameter dialog** - Simulink tool to select and specify *model parameters*. To open the dialog box, in the **Simulation** tab, under **Configuration & Simulation**, click **Model Settings**.
**Performance Advisor** - Simulink tool that checks for conditions that might slow down a simulation. The advisor can automatically make changes to a model to address these conditions or you can review and apply suggested changes manually. To open the advisor, in the **Debug** tab, click **Performance Advisor**.
**Property Inspector** – Simulink tool to define and edit parameters and properties for models and blocks. To open the inspector in a Simulink Editor pane, in the **Modeling** tab, under **Design**, click **Property Inspector**.
**Simulation Data Inspection** – Simulink tool to visualize simulation results and compare results from multiple simulations. Select signals to log by right-clicking a signal and then selecting **Log Selected Signals**. After running a simulation, open the inspector by clicking the toolbar icon 📊.
Upgrade Advisor - Simulink tool to identify new feature benefits and needed changes to your model when upgrading to a new release. To open the advisor, in the Modeling tab, select Model Advisor > Upgrade Advisor.
**debugger, Simulink debugger** – Simulink tool to troubleshoot model simulations by stepping through calls to the block-level methods. To open the debugger, in the **Debug** tab, select **Breakpoints List > Debug Model**.
**Diagnostic Viewer** - Simulink tool that displays error, warning, and information messages for conditions detected during a Simulink operation. To open the viewer, in the *Debug* tab, click *Diagnostics*. 
Library Browser, Simulink Library Browser - Simulink tool for searching and selecting model blocks. To open the browser, click the Simulink Editor toolbar icon.
Model Advisor - Simulink tool that checks a Simulink model for conditions and configuration settings that cause inaccurate or inefficient simulation of a model. To open and run the advisor, in the Modeling tab, click Model Advisor.
Model Browser - Simulink tool to view and navigate the structure of a model. To open the Model Browser in a Simulink Editor pane, in the Modeling tab, select Environment > Model Browser.
Model Data Editor - Simulink tool to define and edit data items in a list that you can sort, group, and filter. For example, enter property and parameter values, such as data types and dimensions without having to locate the items in the block diagram. To open the Model Data Editor, in the Simulink Editor, in the Modeling tab, under Design, click Model Data Editor.
Model Dependency Viewer – Simulink tool to graph the models and libraries referenced directly or indirectly by the parent model. To open the viewer, in the Modeling tab, select Compare > Dependencies > Models & Libraries. In the following example, the plant_model is referenced from the parent cruise_control model.
**Model Explorer** – Simulink tool to view and modify elements of a Simulink model, Stateflow chart, or model workspace variables. To open the explorer window, in the **Modeling** tab, click **Model Explorer**.
Programming Constructs in Simulink

Simulink command - MATLAB command that is specific to Simulink modeling or simulation. Enter Simulink commands in the MATLAB Command Window or use in MATLAB scripts for testing a model programmatically. The following commands set simulation parameters, run a simulation, and saves the simulation results. In the first statement, sim is a Simulink command for running a simulation.

```matlab
simOut = sim('cruise_control','SimulationMode','normal',... 'AbsTol','1e-5','SaveState','on',... 'StateSaveName','xout','SaveOutput','on',... 'OutputSaveName','yout','SaveFormat', 'Dataset');
outputs = simOut.get('yout')
```

callback, model callback – MATLAB code that executes in response to a specific model or block action. To add a model callback, in the Modeling tab, under Design, click Property Inspector, then in the Properties tab, select a function from the Callbacks list, and then enter MATLAB code or the name of a MATLAB script.

In the following example, when Simulink loads a model it also loads a file into the MATLAB workspace with parameters values for the model.
**Simulink function** - Computational unit that calculates a set of outputs when provided with a set of inputs. A common text interface between function caller and function definition allows various definition formats using a Simulink Function block, exported Stateflow graphical function, or exported Stateflow MATLAB function.

**MATLAB Function** -

**S-function, system-function** - Computer language description of a Simulink S-Function block written in MATLAB code, C, C++, or Fortran. C, C++, and Fortran S-functions are compiled as MEX files using the MATLAB `mex` utility.
/ * File : sfunmem.c
  * Abstract:
  * A one integration-step delay and hold "memory" function.
  * Syntax:  [sys, x0] = sfunmem(t,x,u,flag)

#define S_FUNCTION_NAME sfunmem
#include "simstruc.h"

/*==================================================================
 * S-function methods *
 ******************************************************/

/*==================================================================
 * Function: mdlInitializeSizes ========================================
 * Abstract: 
 * Call mdlCheckParameters to verify that the parameters are okay, 
 * then setup sizes of the various vectors. 
 */

static void mdlInitializeSizes(SimStruct *S)
{
    ssSetNumSFcnParams(S, 0); /* Number of expected parameters */
    if (ssGetNumSFcnParams(S) != ssGetSFcnParamsCount(S)) {
        return; /* Parameter mismatch will be reported by Simulink */
    }
}
Model Development Processes

**model-based design** - Development process that uses a system model as an executable specification throughout development. The process supports model and model component design, model simulation of dynamic behavior, code generation from the model, and continuous test and verification.

**software-in-the-loop (SIL) simulation** - Development process where compiled source code on a development computer executes as a separate process from the rest of the Simulink model. Typical goals include initial source code testing and verification by comparing software-in-the-loop results with model simulation results and system requirements.
In the following example, to run a software-in-the-loop simulation for the controller, right-click the Model block labeled control model and then select Block parameters. From the Simulation mode list, select Software-in-the-loop (SIL).

**processor-in-the-loop (PIL) simulation** – Development process that cross-compiles source code and runs the resulting object code on a target processor using hardware-specific data attributes and sample time attributes. Typical goals include object code verification by comparing processor-in-the-loop results with model simulation and software-in-the-loop (SIL) results.

**hardware-in-the-loop (HIL) simulation** – Development process that pairs physical components, such as the hardware and software for a controller, with a virtual real-time implementation of a physical component, such as a plant.
In the following example, code for the controller model is generated and downloaded to the production controller hardware. Code is generated for the plant model and downloaded to a real-time hardware computer (e.g. Simulink Realtime).

**Test Harness**

Simulink model that provides a framework for testing the simulation behavior and outputs of another model or model component. Objectives of the test harness are to automate the testing process, execute a test suite with multiple test cases, and analyze results.

In the following model, a Signal Builder block provides test signals while outputs are logged for analysis in the Simulation Data Inspector or MATLAB.
Block Reference Page Examples

- “Create Bus Ports in a Subsystem” on page 14-7
- “Manage Bus-to-Vector Conversions” on page 14-10
- “Assign Signal Values to a Bus” on page 14-13
- “Initialize Your Model Using the Callback Button Block” on page 14-14
- “Control a Parameter Value with Callback Button Blocks” on page 14-15
- “Control the Duty Cycle of a PWM Signal Using Dashboard Blocks” on page 14-17
- “Control Merging Signals with the Push Button Block” on page 14-19
- “Tune the Relative Slip for an Anti-Lock Braking System” on page 14-22
- “Interactively Simulate a Vehicle Climate Control System” on page 14-24
- “Interactively Simulate a Thermal Model of a House” on page 14-27
- “Create a Realistic Dashboard Using the Custom Gauge Block” on page 14-30
- “Solve a Linear System of Algebraic Equations” on page 14-34
- “Model a Planar Pendulum” on page 14-35
- “Improved Linearization with Transfer Fcn Blocks” on page 14-39
- “View Dead Zone Output on Sine Wave” on page 14-40
- “View Backlash Output on Sine Wave” on page 14-42
- “Prelookup With External Breakpoint Specification” on page 14-44
- “Prelookup with Evenly Spaced Breakpoints” on page 14-45
- “Configure the Prelookup Block to Output Index and Fraction as a Bus” on page 14-46
- “Approximating the sinh Function Using the Lookup Table Dynamic Block” on page 14-48
- “Create a Logarithm Lookup Table” on page 14-50
- “Providing Table Data as an Input to the Direct Lookup Table Block” on page 14-51
- “Specifying Table Data in the Direct Lookup Table Block Dialog Box” on page 14-52
- “Using the Quantizer and Saturation blocks in sldemo_boiler” on page 14-53
• “Scalar Expansion with the Coulomb and Viscous Friction Block” on page 14-54
• “Sum Block Reorders Inputs” on page 14-55
• “Iterated Assignment with the Assignment Block” on page 14-57
• “View Sample Time Using the Digital Clock Block” on page 14-58
• “Bit Specification Using a Positive Integer” on page 14-59
• “Bit Specification Using an Unsigned Integer Expression” on page 14-60
• “Track Running Minimum Value of Chirp Signal” on page 14-61
• “Horizontal Matrix Concatenation” on page 14-63
• “Vertical Matrix Concatenation” on page 14-64
• “Multidimensional Matrix Concatenation” on page 14-65
• “Unary Minus of Matrix Input” on page 14-66
• “Sample Time Math Operations Using the Weighted Sample Time Math Block” on page 14-67
• “Construct Complex Signal from Real and Imaginary Parts” on page 14-68
• “Construct Complex Signal from Magnitude and Phase Angle” on page 14-69
• “Find Nonzero Elements in an Array” on page 14-70
• “Calculate the Running Minimum Value with the MinMax Running Resettable Block” on page 14-71
• “Find Maximum Value of Input” on page 14-73
• “Permute Array Dimensions” on page 14-75
• “Multiply Inputs of Different Dimensions with the Product Block” on page 14-76
• “Multiply and Divide Inputs Using the Product Block” on page 14-77
• “Divide Inputs of Different Dimensions Using the Divide Block” on page 14-78
• “Complex Division Using the Product of Elements Block” on page 14-79
• “Element-Wise Multiplication and Division Using the Product of Elements Block” on page 14-80
• “sin Function with Floating-Point Input” on page 14-81
• “sincos Function with Fixed-Point Input” on page 14-82
• “Trigonometric Function Block Behavior for Complex Exponential Output” on page 14-83
• “Create Bus Object from Constant Block” on page 14-84
• “Control Algorithm Execution Using Enumerated Signal” on page 14-85
• “Integer and Enumerated Data Type Support in the Ground Block” on page 14-87
• “Fixed-Point Data Type Support in the Ground Block” on page 14-88
• “Read 1-D Array and Structure From Workspace” on page 14-89
• “Read Structure From Workspace Using Model Sample Time” on page 14-90
• “Read 2-D Signals in Structure Format From Workspace” on page 14-92
• “From File Block Loading Timeseries Data” on page 14-93
• “Eliminate Singleton Dimension with the Squeeze Block” on page 14-94
• “Difference Between Time- and Sample-Based Pulse Generation” on page 14-95
• “Specify a Waveform with the Repeating Sequence Block” on page 14-97
• “Tune Phase Delay on Pulse Generator During Simulation” on page 14-99
• “Difference Sine Wave Signal” on page 14-100
• “Discrete-Time Derivative of Floating-Point Input” on page 14-102
• “First-Order Sample-and-Hold of a Sine Wave” on page 14-104
• “Calculate and Display Simulation Step Size using Memory and Clock Blocks” on page 14-106
• “Capture the Velocity of a Bouncing Ball with the Memory Block” on page 14-107
• “Implement a Finite-State Machine with the Combinatorial Logic and Memory Blocks” on page 14-109
• “Discrete-Time Integration Using the Forward Euler Integration Method” on page 14-112
• “Signal Routing with the From, Goto, and Goto Tag Visibility Blocks” on page 14-113
• “Zero-Based and One-Based Indexing with the Index Vector Block” on page 14-116
• “Noncontiguous Values for Data Port Indices of Multiport Switch Block” on page 14-117
• “Using Variable-Size Signals on the Delay Block” on page 14-118
• “Bus Signals with the Delay Block for Frame-Based Processing” on page 14-120
• “Control Execution of Delay Block with Enable Port” on page 14-121
• “Zero-Based Indexing for Multiport Switch Data Ports” on page 14-123
• “One-Based Indexing for Multiport Switch Data Ports” on page 14-124
• “Enumerated Names for Data Port Indices of the Multiport Switch Block” on page 14-126
• “Prevent Block Windup in Multiloop Control” on page 14-127
• “Bumpless Control Transfer” on page 14-128
• “Bumpless Control Transfer with a Two-Degree-of-Freedom PID Controller” on page 14-129
• “Using a Bit Set block” on page 14-130
• “Using a Bit Clear block” on page 14-131
• “Two-Input AND Logic” on page 14-132
• “Circuit Logic” on page 14-133
• “Unsigned Inputs for the Bitwise Operator Block” on page 14-134
• “Signed Inputs for the Bitwise Operator Block” on page 14-135
• “Merge Block with Input from Atomic Subsystems” on page 14-136
• “Index Options with the Selector Block” on page 14-137
• “Switch Block with a Boolean Control Port Example” on page 14-139
• “Merge Block with Unequal Input Widths Example” on page 14-140
• “Detect Rising Edge of Signals” on page 14-143
• “Detect Falling Edge Using the Detect Fall Nonpositive Block” on page 14-145
• “Detect Increasing Signal Values with the Detect Increase Block” on page 14-147
• “Extract Bits from Stored Integer Value” on page 14-149
• “Detect Signal Values Within a Dynamically Specified Interval” on page 14-150
• “Model a Digital Thermometer Using the Polynomial Block” on page 14-152
• “Convert Data Types in Simulink Models” on page 14-153
• “Control Data Types with the Data Type Duplicate Block” on page 14-155
• “Probe Sample Time of a Signal” on page 14-156
• “Convert Signals Between Continuous Time and Discrete Time” on page 14-157
• “Convert Muxed Signal to a Vector” on page 14-159
• “Create Contiguous Copy of a Bus Signal” on page 14-160
• “Convert Virtual Bus to a Nonvirtual Bus” on page 14-161
• “Convert Nonvirtual Bus to Virtual Bus” on page 14-162
• “Remove Scaling from a Fixed-Point Signal” on page 14-163
• “Stop Simulation Block with Relational Operator Block” on page 14-164
• “Output Simulation Data with Blocks” on page 14-165
• “Increment and Decrement Real-World Values” on page 14-170
• “Increment and Decrement Stored Integer Values” on page 14-173
• “Specify a Vector of Initial Conditions for a Discrete Filter Block” on page 14-174
• “Generate Linear Models for a Rising Edge Trigger Signal” on page 14-176
• “Generate Linear Models at Predetermined Times” on page 14-178
• “Capture Measurement Descriptions in a DocBlock” on page 14-180
• “Square Root of Negative Values” on page 14-181
• “Signed Square Root of Negative Values” on page 14-182
• “rSqrt of Floating-Point Inputs” on page 14-183
• “rSqrt of Fixed-Point Inputs” on page 14-184
• “Model a Series RLC Circuit” on page 14-185
• “Extract Vector Elements and Distribute Evenly Across Outputs” on page 14-188
• “Extract Vector Elements Using the Demux Block” on page 14-189
• “Detect Change in Signal Values” on page 14-191
• “Detect Fall to Negative Signal Values” on page 14-193
• “Detect Decreasing Signal Values” on page 14-195
• “Function-Call Blocks Connected to Branches of the Same Function-Call Signal” on page 14-197
• “Function-Call Feedback Latch on Feedback Signal Between Child and Parent” on page 14-198
• “Single Function-Call Subsystem” on page 14-199
• “Function-Call Subsystem with Merged Signal As Input” on page 14-200
• “Partitioning an Input Signal with the For Each Block” on page 14-201
• “Specifying the Concatenation Dimension in the For Each Block” on page 14-202
• “Working with the Initialize Function, Reset Function, and Terminate Function Blocks” on page 14-203
• “Reading and Writing States with the Initialize Function and Terminate Function Blocks” on page 14-204
<xi:include href="urn:mathworks:ex:simulink/VariableSpeedConveyorBeltExample.xml"
xmlns:xi="http://www.w3.org/2001/XInclude">

<xifallback><!-- Fallback for simulink/VariableSpeedConveyorBeltExample -->
<link xmlns="http://docbook.org/ns/docbook"/>
</xifallback>

</xi:include>
Create Bus Ports in a Subsystem

This example shows adding In Bus Element and Out Bus Element blocks to create bus element ports in a subsystem for selecting signals from an input bus and creating an output bus signal.

Model Structure

Open the model. The top model has three constant signals combined into a bus that feeds a subsystem input port and outputs a bus signal to a Scope block.

The subsystem includes two In Bus Element and Out Bus Element blocks.
Set the Port Name and Color for an In Bus Element Block

Open the Block Parameters dialog box for the ConstIn.a block. The Port name parameter is set to ConstIn, which changes the subsystem input port name from the default InBus. This block selects the a signal. The block color is set to green instead of the default black.

The In Bus Element block feeds a Gain block, and the Gain block connects to an Out Bus Element block that includes an a*3 signal in the output bus signal.

Create an In Bus Element Block and an Out Bus Element Block

Create a third In Bus Element block for the c signal. In the Simulink Editor, right-click and drag the ConstIn.b block to make a copy of the block. Specify to use the existing port. Edit the block icon text to say ConstIn.c. Select the block and override the green block color by specifying black. Feed the output of the In Bus Element block to a third Gain block and set the gain to 10.

Copy the OutBus.b*5 block to create another Out Bus Element block, specifying to use the same port. Connect the Gain block to the Out Bus Element block that uses the same port. Connect the Gain block output signal to the Out Bus Element block and edit the icon text to say OutBus.c*10.
Add a Chirp Block and Include Its Output Signal in the Bus Output Signal

Add a Chirp Signal block and connect it to a new Out Bus Element block that uses the same port as the other Out Bus Element blocks (OutBus). Open the Out Bus Element dialog box, double-click the selected signal, and change the signal name to chirp.

The virtual bus signal that the subsystem outputs contains the output signals of the three Gain blocks and the Chirp block.
Manage Bus-to-Vector Conversions

This example shows how to find and manage implicit bus-to-vector conversions.

Blocks that do not accept buses may implicitly convert buses to vectors. When a bus is treated as a vector, bus elements become inaccessible.

Some buses cannot convert to vectors. For more information, see Bus to Vector.

**Identify Implicit Bus-to-Vector Conversions**

Open and simulate model **ex_bus_to_vector**.

![Diagram](image)

To accept the bus, the Gain blocks implicitly convert the bus to a vector.

To identify buses treated as vectors before simulation, use function `Simulink.BlockDiagram.addBusToVector`.

```
[blocks] = Simulink.BlockDiagram.addBusToVector('ex_bus_to_vector')
```

```matlab
### Processing block diagram 'ex_bus_to_vector'
### Number of blocks left that are connected to a bus being used as a vector: 2
### Done processing block diagram 'ex_bus_to_vector'
```

```
blocks =

1x2 struct array with fields:
   BlockPath
   InputPort
   LibPath
```
To identify buses treated as vectors during simulation, set the **Bus signal treated as vector** configuration parameter to **warning** or **error**. The default setting for **Bus signal treated as vector** is **none**, which generates no warning or error message when a block implicitly converts a bus to a vector.

**Explicitly Define Bus-To-Vector Conversions**

To insert Bus to Vector blocks where blocks implicitly convert buses to vectors, use function `Simulink.BlockDiagram.addBusToVector` with `reportOnly` set to `false`. When you use function `Simulink.BlockDiagram.addBusToVector` with `reportOnly` set to `false`, the function saves the model. To create a writable copy of model `ex_bus_to_vector`, this example uses the `save_system` function.

```matlab
save_system('ex_bus_to_vector','ex_bus_to_vector_blocks');
[blocks,busToVectors] = Simulink.BlockDiagram.addBusToVector('ex_bus_to_vector_blocks',true,false);
```

The Gain blocks no longer implicitly convert the bus to a vector. The inserted Bus to Vector blocks perform the conversion explicitly.

Bus to Vector blocks are virtual and do not affect simulation results, code generation, or performance.

Function `Simulink.BlockDiagram.addBusToVector` returns no remaining implicit bus-to-vector conversions.

```matlab
[blocks] = Simulink.BlockDiagram.addBusToVector('ex_bus_to_vector_blocks')
```

### No buses used as vectors left to process

blocks =
1x0 empty struct array with fields:

BlockPath
InputPort
MixedAttributes

By specifying acceptable bus-to-vector conversions with Bus to Vector blocks, you can more easily identify unexpected conversions. Having configuration parameter **Bus signal treated as vector** set to **warning** or **error** alerts you when an unexpected bus-to-vector conversion occurs.
Assign Signal Values to a Bus

This example shows how to use a Bus Assignment block to change a bus element value without adding Bus Selector and Bus Creator blocks to select bus elements and reassemble them into a bus.

Open the model and simulate it.

Initially, the value of signal a is 1. However, the Bus Assignment block replaces that initial value of signal a with the value of signal c, which is 3. The const_bus output signal has a value of 3 for signal a, as the Display block shows.
Initialize Your Model Using the Callback Button Block

This example shows how to use the Callback Button block to perform initialization routines on your model.

Explore the Model

The example model builds on the sldemo_fuelsys featured model. When you open the model, to bind the workspace variables to their Dashboard blocks, you have to update the model diagram. Here, the Callback Button block at the bottom of the Dashboard subsystem in the model has been configured to update the diagram on the release of the mouse button when you click the block.

Fault-Tolerant Fuel Control System Dashboard

You do not need to start a simulation for the Callback Button to react to your input. Just select and then click the Callback Button to run the initialization code. Double-click the Callback Button block to view and edit its parameters, including the press and click scripts.
Control a Parameter Value with Callback Button Blocks

This example models control of a system that consists of two masses attached on either side of a spring. A control loop damps the oscillation of the spring that results when an external force acts on the system. The model uses Callback Button blocks to provide an interface for you to adjust the frequency of the external force before and during simulation.

Explore the Model

The model for this example adds two Callback Button blocks, labeled Frequency + and Frequency - to the Double Mass-Spring System model. When you simulate the model, an animation visualizes the system.

Click the button labeled Frequency + to increase the oscillation frequency. When you adjust the frequency of the external force, the Callback Button block displays a message.
in the command window indicating the new frequency value. You can adjust the parameter during a simulation and while the model is idle.

Both Callback Button blocks in this model are configured with a ClickFcn that responds to your clicks and a PressFcn that executes when you press the Callback Button block. Double-click the Frequency + Callback Button block to view its parameters.

When you click the Frequency + Callback Button block, the ClickFcn increases the frequency of the external force by 0.1. If you press the Callback Button block for more than the 500 ms Press Delay, the PressFcn increases the frequency of the external force by 0.1 every second.
Control the Duty Cycle of a PWM Signal Using Dashboard Blocks

This example shows how a simulation dashboard built using Dashboard blocks controls the duty cycle of a PWM signal generated by the model. The example is a modified version of the sfcndemo_pwm example. The PWM Generator block takes a duty cycle and period as inputs to generate a PWM output. You can control the source for the duty cycle using a switch.

Build the Dashboard Subsystem

To create the Dashboard subsystem in this example:

1. Add a Subsystem block to the model.

2. Delete the pre-populated contents of the subsystem.
3. Add the contents of the dashboard. The dashboard in this example includes a Rocker Switch block and a Knob block, which allow you to modify variables during simulation. The Dashboard Scope block displays the generated PWM signal.

Use areas in the Dashboard subsystem to group and label Dashboard blocks. For more information about creating areas, see “Box and Label Areas of a Model”.

**Connect Blocks in the Model**

In this model, the Rocker Switch block in the Dashboard subsystem provides the interface for selecting the duty cycle source. A Switch block with a Constant block as its control input replaces the Manual Switch in the sfcndemo_pwm model. Replacing the Manual Switch block requires three blocks but allows you to gather model controls and visualizations in a single view. The Rocker Switch block connects to the value of the Constant block. In this model, the Constant block value is set by the variable `duty_cycle_source`.

This model also uses the variable `duty_cycle` to set the magnitude of the duty cycle for both the duty cycle (constant) and duty cycle (varying) sources. The Knob block in the Dashboard subsystem connects to the `duty_cycle` variable, allowing you to modify its value during simulation.

The Dashboard Scope block displays the output of the PWM Generator block, allowing you to observe the generated PWM signal during simulation.

**Interactively Simulate**

This model uses simulation pacing to slow model execution so you can interact with the model during simulation. For more information, see “Simulation Pacing”.

Run the simulation. Use the Rocker Switch block to select the varying duty cycle input and observe the resulting PWM waveform on the Dashboard Scope block. Change the magnitude of the duty cycle using the Knob block and observe the effect on the generated PWM signal.
Control Merging Signals with the Push Button Block

This example model is a modified version of the “Merging Signals” on page 15-407 example. The `mergedemo` model illustrates how to use Simulink® to merge two different signals using source blocks and triggered subsystems. In the `mergedemo` model, a square wave enables and disables the triggered subsystems that provide input to the Merge block. In this model, you can control the enable signal for the triggered subsystems using the Signal Select Push Button block.

Model Modifications

To create this model from the `mergedemo` example:

1. Delete the Discrete Pulse Generator block controlling the triggered subsystem and the Scope block.

Copyright 2004-2018 The MathWorks, Inc.
2. Add a Push Button block, an Outport block, a Constant block, and a Dashboard Scope block.

3. Connect the Outport block to the output of the Merge block.

4. Display the output signal of the Merge block on the Dashboard Scope.

5. Connect the Constant block to the input of the Data Type Conversion block and specify a variable name as its value. This model uses the variable name `signal_select`. Click the Unrecognized functions or variables error, and create the variable in the Model Workspace.

![Unrecognized functions or variables](image)

Now, the variable `signal_select` controls the triggered subsystems.

6. Press Ctrl+D to update the diagram so the variable you just created is available for connection.

7. Connect the Push Button block to the `signal_select` variable.

   In this example, the On Value for the Push Button block is set to 0. When you press the Push Button block, the value of `signal_select` changes to 0, enabling the triggered subsystem connected to the sawtooth wave. If you want pressing the Push Button block to enable the triggered subsystem connected to the sine wave, set the On Value for the Push Button block to 1.

**Interactive Simulation**

This model uses simulation pacing to slow model execution so you can interact with the model during simulation. For more information about simulation pacing, see “Simulation Pacing”.
Run the simulation. As the model simulates, press and hold the Signal Selection Push Button block. You can observe the effect of pressing the Push Button block on the Dashboard Scope block.
Tune the Relative Slip for an Anti-Lock Braking System

This example shows how to use a subsystem made of Dashboard blocks to tune the value of the Desired relative slip for the “Modeling an Anti-Lock Braking System” on page 15-210 example. You can use the Slider block or the Edit block to change the value of the Desired relative slip and observe the results during simulation on the Dashboard Scope block.

Build the Dashboard Subsystem

This model uses a Subsystem block to gather the controls and displays for the model in a single interface. To create the subsystem for this model:

1. Add a Subsystem block to the sldemo_absbrake model.

2. Delete the pre-populated contents of the subsystem. A subsystem of Dashboard blocks does not need Inport or Outport blocks because Dashboard blocks do not use ports to make connections.

3. Add the components of the dashboard.
The dashboard for this model includes an Edit block and a Slider block that both control the value of the Desired relative slip input. The Dashboard Scope block displays the resulting slip value, slp throughout the simulation.

You can use areas in the subsystem to group and label related controls and displays. For more information about creating areas, see “Box and Label Areas of a Model”.

**Interactively Simulate**

You can use the Dashboard subsystem as a convenient interface to configure separate simulations that use different values for the Desired relative slip.

The Slider block and the Edit block both change the value of the Desired relative slip input. Use the Slider block to fine tune the value of the relative slip input, and use the Edit block to enter an exact value.
Interactively Simulate a Vehicle Climate Control System

In this example, you use a Dashboard of controls and displays to interactively simulate a climate control system of a car. You can adjust the fan and air recycling controls as well as set the internal temperature and modify the external temperature. A Linear Gauge block, a Dashboard Scope block, and a Display block show the internal temperature of the car. The model in this example adds the Dashboard to the sldemo_auto_climatecontrol model. For more information about the underlying model, see “Simulating Automatic Climate Control Systems” on page 15-242.

Build the Dashboard Subsystem

This model uses a Subsystem block to gather the controls and displays in a single interface for the model. To create the subsystem in this example:

1. Add a Subsystem block to the sldemo_auto_climatecontrol model.
2. Delete the prepopulated contents of the subsystem. The subsystem does not need inports or outports because the Dashboard blocks do not use ports to make connections.

3. Add the contents of the dashboard.

In the dashboard for this model:

- Two Check Box blocks turn the fans and air recycling on and off.
- A Knob block sets the internal temperature.
- An Edit block allows you to specify the external temperature.
- A Dashboard Scope block displays the value of the internal temperature over time.
- A Linear Gauge block displays the instantaneous internal temperature value.
- A Display block shows the instantaneous internal temperature value.

You can use areas in the subsystem to group and label related controls and displays. For more information about creating areas, see “Box and Label Areas of a Model”.

**Connect Blocks in the Model**

To explore connections between each block in the Dashboard subsystem and the model, you can select a block in the dashboard, pause on the ellipsis that appears above it, and click Jump to Connected Element. The Simulink® editor navigates to and highlights the block or signal connected to that block in the dashboard.

The Check Box blocks connect to the `dist_req` and `recycle_air` constant blocks. In the `sldemo_auto_climatecontrol` model, the fan and air recycling signals are controlled by Manual Switch blocks with two Constant blocks as inputs.

To replace a Manual Switch block in the `sldemo_auto_climatecontrol` model with a Check Box block:

1. Delete the Manual Switch block and one of the Constant blocks.
2. Connect the other Constant block directly to the Mux block.
3. Specify a variable as the Constant block value.

This model uses the variable `dist_req` to control the fans and the variable `recycle_air` to control air recycling.
4. Click the **Unrecognized functions or variables** error, and create the variable in the Model Workspace.

Now, the variable `dist_req` holds the state of the fans.

5. Press Ctrl+D to update the diagram so the variable you just created is available for connection.

6. Connect the Check Box block to the `dist_req` variable.

Follow a similar process to replace the air recycling manual switch.

The Knob block connects to the `USet` parameter of the User Setpoint in Celsius masked subsystem. Similarly, the Edit block connects to the `x` parameter of the External Temperature in Celsius masked subsystem. The Dashboard Scope, Linear Gauge, and Display blocks all connect to the output signal of the Kelvin to Celsius subsystem.

**Interactive Simulation**

This model uses simulation pacing to slow model execution so you can interact with the model during simulation. For more information, see “Simulation Pacing”.

Run the simulation. Use the Check Box blocks to control the fans and air recycling. Use the Knob block to adjust the internal temperature set point, and specify the external temperature with the Edit block. You can observe the resulting internal temperature on the Dashboard Scope block, the Linear Gauge block, and the Display block.
Interactively Simulate a Thermal Model of a House

This example uses a dashboard of controls and indicators to configure and then interactively simulate a thermal model of a house. You can configure the simulation to use either traditional or renewable energy sources and to use an outdoor temperature setpoint based on the season. During the simulation, you can adjust the thermostat and monitor the indoor temperature and heat cost. The model in this example adds a subsystem named Dashboard to the sldemo_househeat model. For more information about the underlying model, see “Thermal Model of a House” on page 15-31.

Build the Dashboard Subsystem

This model uses a Subsystem block to organize the controls and displays for the model in a single interface. To create the subsystem in this example:

1. Add a Subsystem block to the model.
2. Delete the prepopulated contents of the subsystem. A subsystem of Dashboard blocks does not require Inport or Outport blocks because Dashboard blocks do not use ports to make connections.

3. Add the components of the dashboard.

The dashboard for this model includes a Knob block, a Radio Button block, and a Combo Box block to modify simulation parameters. A Linear Gauge block, three Display blocks, and a Dashboard Scope block display key signals during simulation.

You can use areas in the subsystem to group and label related controls and displays. For more information about creating areas, see “Box and Label Areas of a Model”.

**Connect Dashboard Blocks to the Model**

To explore connections between each block in the Dashboard subsystem and the model, you can select a block in the dashboard, pause on the ellipsis that appears above it, and click Jump to Connected Element. The Simulink® editor navigates to and highlights the block or signal connected to that block in the dashboard.

- The Knob block connects to the Set Point block and allows you to adjust the thermostat set point in the model during simulation.
- The Radio Button Block connects to the Avg Outdoor Temp block and maps seasonal average temperatures for Boston, MA, to season labels. Right-click the Radio Button block and select Block Parameters from the context menu to modify the average temperature value used for each season.
- The Display block in the Avg Outdoor Temperature area shows the outdoor temperature value used in the simulation based on the selected season.
- The Combo Box block connects to the Gain block used to calculate the cost during simulation. Right-click the Combo Box block and select Block Parameters from the context menu to view or modify the cost, specified in units of cents per Joule.
- The Linear Scale and Display blocks in the Indoor Temperature area provide analog and digital indications of the instantaneous value of the house temperature during simulation.
- The Display block in the Heat Cost area shows the cumulative cost of heating the house over the course of the simulation is shown on a Display block.
- The Dashboard Scope block displays the heat cost, indoor temperature, and outdoor temperature in the same plot.
Interactively Simulate

This model uses simulation pacing to slow model execution so you can interact with the model during simulation. For more information, see “Simulation Pacing”.

The simulation is set up to run for 48 hours. Within that time period, it makes sense to use the Radio Button block and Combo Box block to configure a simulation for a particular season and energy source. If you want to run another type of simulation where you interactively modify the season or energy source, consider lengthening the simulation time to something more realistic for those types of changes.

Choose the season and energy source for a 48-hour simulation, and then run the simulation. During the simulation, use the Knob block to adjust the set point for the house thermostat, and observe the effect of the change on the indoor temperature and heat cost.
Create a Realistic Dashboard Using the Custom Gauge Block

You can use the Custom Gauge block to create a dashboard of controls and indicators for your model that looks how it would in a real system.

Model Overview

This example model uses four Custom Gauge blocks and a MultiStateImage block to create a dashboard for the sf_car model like one you might see in a real car.

Explore the Model

To understand the connections between the Dashboard and model, you can select each block and jump to the signal it displays in the model. To jump to the connected signal, hover on the ellipsis that appears over the block when you select it. Then, click the arrow button.

The Custom Gauge block on the left represents the vehicle speed signal, acting as a speedometer on the dashboard and using the default Custom Gauge appearance and a value arc you can see when you simulate the model.
The Custom Gauge blocks at the top of the model display the throttle and brake signals. You can create this type of gauge by deleting the background image and needle image from the default Custom Gauge. Then, draw an arc and select colors for the tick marks, arc, and value arc.

The Custom Gauge block on the right displays the engine RPM signal on a custom gauge face. You can create a gauge like this by deleting the background image in the default Custom Gauge and uploading your own.
The MultiStateImage block on the bottom displays the gear signal using seven segment display style number images that correspond to the value of the gear signal.

Observe System During Simulation

You can use the dashboard to monitor the system response during simulation. The User Inputs subsystem includes several simulation inputs to model different vehicle maneuvers. To change the simulation input, navigate to the top level of the model and double-click the User Inputs subsystem. You can use the drop-down menu at the top of the gui to choose the vehicle action for the model to simulate. Choose between a passing maneuver, gradual acceleration, hard braking, and coasting. To monitor the system
response during simulation, navigate back to the Dashboard subsystem and then press the play button to simulate the model.
Solve a Linear System of Algebraic Equations

Use the Algebraic Constraint block to solve the system

\[
\begin{align*}
z_1 + z_2 &= 1 \\
z_2 - z_1 &= 1
\end{align*}
\]

The model represents the problem in a vectorized form as

\[
\begin{bmatrix}
1 & 1 \\
-1 & 1
\end{bmatrix}
\begin{bmatrix}
z_1 \\
z_2
\end{bmatrix}
= 
\begin{bmatrix}
1 \\
1
\end{bmatrix}
\]

The signal fed to the Algebraic Constraint block \( f(z) \) is a \( 2 \times 1 \) vector of the form

\[
\begin{bmatrix}
z_1 + z_2 - 1 \\
-z - 1 + z_2 - 1
\end{bmatrix}
\]

The block is configured to constrain \( f(z) \) to 0. Thus solving for \( f(z) = 0 \) yields the solution \( z_1 = 0, \ z_2 = 1 \)
Model a Planar Pendulum

Consider a point mass $m$ suspended by a massless rod of length $l$ under the influence of gravity. The position of the mass can be expressed in Cartesian coordinates by $(x,y)$.

Modeling the System

A force balance of the mass gives the equations of motion in the $x$ and $y$ directions.

\[ m \ddot{x} = F \sin \theta \quad (1) \]
\[ m \ddot{y} + F \cos \theta = -mg \quad (2) \]

Let $(u, v)$ be the velocities in $(x, y)$ respectively. The system can be rewritten as a system of first order ODEs.
\[
\begin{align*}
\dot{x} &= u \\
\dot{u} &= -F \frac{x}{ml} \\
\dot{y} &= v \\
\dot{v} &= -F \frac{y}{ml} - g
\end{align*}
\] (3) (4) (5) (6)

where \( F \) is the tension in the rod. The system also possesses the geometric constraint
\[ x^2 + y^2 = l^2 \] (7)

Differentiate (7) twice with respect to time \( t \) to arrive at
\[ m(u^2 + v^2) - Fl - mgy = 0 \] (8)

This relationship is useful since it allows \( F \) to determined at every step for use in modeling the kinematics of the system.

**Simulating the System**

The system is simulated as shown in the figure below

Equation (8) contains one unknown \( F \) and is of the form \( f(z) = 0 \) where \( f(z) = m(u^2 + v^2) - Fl - mgy \). The Algebraic Constraint block constrains \( f(z) \) to 0 and solves for \( F \) in accordance with (8).
Position and velocity in $x$ over time

$$(x, u)$$

$$(t)$$

$$(x)$$

$$(u)$$
References

Improved Linearization with Transfer Fcn Blocks

The Laplace domain transfer function for the operation of differentiation is:

\[
\frac{Y(s)}{X(s)} = s
\]

This equation is not a proper transfer function, nor does it have a state-space representation. As such, the Simulink software linearizes this block as an effective gain of 0 unless you explicitly specify that a proper first-order transfer function should be used to approximate the linear behavior of this block.

To improve linearization, you can also try to incorporate the derivative term in other blocks. For example, if you have a Derivative block in series with a Transfer Fcn block, try using a single Transfer Fcn block of the form

\[
\frac{s}{s + a}
\]

For example, you can replace the first set of blocks in this figure with the blocks below them:

![Diagram of improved linearization with Transfer Fcn Blocks](image-url)
View Dead Zone Output on Sine Wave

This example shows the effect of the Dead Zone block on a sine wave. The model uses a dead zone lower limit of -0.5 and an upper limit as 0.5. Set these values through the parameters Start of Dead Zone and End of Dead Zone.
View Dead Zone Output on Sine Wave
View Backlash Output on Sine Wave

This example shows the effect of the Backlash block on a sine wave using default parameters. The initial **Deadband width** is 1 and the **Initial output** is 0.

The initial deadband is centered around 0 and has a width of 1, which extends .5 in each direction. The output from the Backlash block begins at 0 and does not change until the input reaches the edge of the deadzone at .5. Then the output engages in a positive direction and changes an equal amount as the input. After the input reaches a value of 1, it starts moving in a negative direction. At this point the output disengages and stays flat.
until the input passes through the deadband width of 1. Once the input reaches the end of the deadband zone at $\theta$, then the output engages and starts moving in a negative direction with the input.
Prelookup With External Breakpoint Specification

This example shows how to feed a breakpoint dataset from a Constant block to the bp input port of the Prelookup block.

The Prelookup block inherits the following breakpoint attributes from the bp input port:

- **Minimum**: Inf
- **Maximum**: Inf
- **Data type**: single

Similarly, a Constant block feeds the table data values to the T input port of the Interpolation Using Prelookup block, which inherits the following attributes:

- **Minimum**: Inf
- **Maximum**: Inf
- **Data type**: single

Simulink® uses double-precision, floating-point data to perform the computations in this model. However, the model stores the breakpoint and table data as single-precision, floating-point data. Using a lower-precision data type to store breakpoint and table data reduces the memory requirement.
Prelookup with Evenly Spaced Breakpoints

This example shows how to specify evenly spaced breakpoint data in the Prelookup block.

In the Breakpoints data section, the Specification parameter is set to Even spacing. The parameters First point, Spacing, and Number of points are set to 25, 12, and 4 respectively. Specifying these parameters creates four evenly spaced breakpoints: [25, 37, 49, 61].

An alternative way to specify evenly spaced breakpoints is to set Specification to Explicit values and set Value to [25:12:61].

Simulink® uses double-precision, floating-point data to perform the computations in this model. However, the model stores the breakpoints and table data as double.
Configure the Prelookup Block to Output Index and Fraction as a Bus

This example shows how to output a bus containing the index \( (k) \) and fraction \( (f) \) from the Prelookup block. The bus object can then be used as an input to the Interpolation Using Prelookup block. The example also shows how to get the same results without using a bus object.

Open and simulate the model.

```
ans =
Simulink.SimulationOutput:
    tout: [11x1 double]
    yout: [1x1 Simulink.SimulationData.Dataset]
SimulationMetadata: [1x1 Simulink.SimulationMetadata]
ErrorMessage: [0x0 char]
```
At the top of the model, open the dialog box for the Prelookup block. In the **Main** tab, note that **Output selection** is set to **Index and fraction as bus**. In the **Data Types** tab, note that **Output** is set to **Bus**: myBus. In the Simulink® Editor, select **File>Model Properties** and open the **Callbacks** tab. In the model’s **PreLoadFcn**, the code defines the bus object myBus, which specifies the index as the first bus element and the fraction as the second element.

Open the dialog box for the Interpolation Using Prelookup block. In the **Main** tab, note that **Require index and fraction as bus** check box is selected. That option configures the block to use the bus output from the Prelookup block.
Approximating the sinh Function Using the Lookup Table Dynamic Block

This example shows how to use the Lookup Table Dynamic block to approximate the sinh function. The breakpoint data is given by the vector [-5:5] and the table data is given by the vector sinh([-5:5]). The input x is provided by the Constant block as a 1-by-3 vector containing values that are below, within, and above the breakpoint data values.

To see how each lookup method handles input values that are below, within, and above the breakpoint data values, change the value of the Lookup Method parameter on the Lookup Table Dynamic block.

The Lookup Table Dynamic block outputs the following values when using the specified lookup methods and inputs.
### Approximating the sinh Function Using the Lookup Table Dynamic Block

<table>
<thead>
<tr>
<th>Lookup Method</th>
<th>Input</th>
<th>Output</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interpolation-Extrapolation</td>
<td>1.4</td>
<td>2.156</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td>5.2</td>
<td>83.59</td>
<td>N/A</td>
</tr>
<tr>
<td>Interpolation-Use End Values</td>
<td>1.4</td>
<td>2.156</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td>5.2</td>
<td>74.2</td>
<td>The block uses the value for sinh(5.0).</td>
</tr>
<tr>
<td>Use Input Above</td>
<td>1.4</td>
<td>3.627</td>
<td>The block uses the value for sinh(2.0).</td>
</tr>
<tr>
<td></td>
<td>5.2</td>
<td>74.2</td>
<td>The block uses the value for sinh(5.0).</td>
</tr>
<tr>
<td>Use Input Below</td>
<td>1.4</td>
<td>1.175</td>
<td>The block uses the value for sinh(1.0).</td>
</tr>
<tr>
<td></td>
<td>-5.2</td>
<td>-74.2</td>
<td>The block uses the value for sinh(-5.0).</td>
</tr>
<tr>
<td>Use Input Nearest</td>
<td>1.4</td>
<td>1.175</td>
<td>The block uses the value for sinh(1.0).</td>
</tr>
</tbody>
</table>
Create a Logarithm Lookup Table

This example shows how to use the n-D Lookup Table block to create a logarithm lookup table. The lookup table allows you to approximate the common logarithm (base 10) over the input range [1,10] without performing an expensive computation.
Providing Table Data as an Input to the Direct Lookup Table Block

This example shows how to provide table data as an input to the Direct Lookup Table block. In the following model, \( a \) is a 4-D array of linearly increasing values that you define with the following model preload function:

\[
a = \text{reshape}(1:2800, [4 5 20 7]);
\]

When you run the model, you get the following results:

The block labeled TableData feeds a 4-D array to the Direct Lookup Table (n-D) block, with a data type of double. Because the Direct Lookup Table (n-D) block uses zero-based indexing, the output is:

\[
a(:,2,4,3)
\]

The output has the same data type as the table data input signal: double.
Specifying Table Data in the Direct Lookup Table Block Dialog Box

This example shows how to specify table data on the dialog box of the Direct Lookup Table (n-D) block. In the following model, the table data is a is a 4-D array of linearly increasing values that you define with the following model preload function:

```m
a = reshape(1:2800, [4 5 20 7]);
```

When you run the model, you get the following results:

Because the Direct Lookup Table (n-D) block uses zero-based indexing, the output is:

```m
a(:,2,4,3)
```

The output data type matches the Direct Lookup Table block's **Table data type**, which is set to **int16**.
Using the Quantizer and Saturation blocks in sldemo_boiler

This example shows how the Quantizer and Saturation blocks are used in the model ex_sldemo_boiler. The ADC subsystem digitizes the input analog voltage by:

- Multiplying the analog voltage by 256/5 with the Gain block
- Rounding the value to integer floor with the Quantizer block
- Limiting the output to a maximum of 255 (the largest unsigned 8-bit integer value) with the Saturation block

This subsystem models a typical 8-bit analog-to-digital converter having an input range of 0-5 volts.
Scalar Expansion with the Coulomb and Viscous Friction Block

This example shows a model with a scalar input to a Coulomb & Viscous Friction block that uses scalar expansion to output a vector.

Double click the friction block to see the parameters. **Coefficient of viscous friction (Gain)** is a scalar value 2, but **Coulomb friction value (Offset)** is a vector value $[1 \ 3 \ 2 \ 0]$. Therefore, the block uses element-wise scalar expansion to compute the output.

Each output is calculated using this formula.

$$ y = \text{sign}(x) \cdot (\text{Gain} \cdot \text{abs}(x) + \text{Offset}) $$

For example, the first offset 1 is calculated as follows.

$$ y = - \cdot ((2 \cdot 5) + 1) $$

$$ y = -11 $$

If the dimensions for the input and Offset are the same, then no expansion is necessary.
Sum Block Reorders Inputs

This example shows how the Sum block reorders inputs. If you use a - sign as the first operation, the block reorders the inputs, if possible, to use a + operation. For example, in the expression output = -a - b + c, the Sum block reorders the input so that output = c - a - b. To initialize the accumulator, the Sum block uses the first + input port.

The block avoids performing a unary minus operation on the first operand a because doing so can change the value of a for fixed-point data types. In that case, the output value differs from the result of accumulating the values for a, b, and c.

Both the constant inputs use int8 data types The Sum block also uses int8 for the accumulator and output data types and has Saturate on integer overflow turned on. The Sum block reorders the inputs to give the ideal result of 127.

1 Reorders inputs from (-Input1 + Input2) to (Input2 - Input1).
2 Initializes the accumulator by using the first + input port. Accumulator = int8(-1) = -1
3 Continues to accumulate values. Accumulator = Accumulator - int8(-128) = 127
4 Calculates the block output. Output = int8(127) = 127

If the Sum block does not reorder the inputs, then you get the nonideal result of 126.

1 Initializes the accumulator by using the first input port. Accumulator = int8((-128)) = 127
2 Because saturation is on, the initial value of the accumulator saturates at 127 and does not wrap.
3 Continues to accumulate values. Accumulator = Accumulator + int8(-1) = 126
4 Calculates the block output. $\text{Output} = \text{int8}(126) = 126$

To explicitly specify a unary minus operation for $\text{output} = -a - b + c$, you can use the Unary Minus block in the Math Operations library.
Iterated Assignment with the Assignment Block

This example shows using the Assignment block to assign values computed in a For or While Iterator loop to successive elements. You can use vector, matrix or multidimensional signals and do the assignment in a single time step. In this model, the For Iterator block creates a vector signal each of whose elements equals $3 \times i$ where $i$ is the index of the element.

The iterator generates indices for the Assignment block. On the first iteration, the Assignment block copies the first input (Y0) to the output (Y) and assigns the second input (U) to the output Y(E1). On successive iterations, the Assignment block assigns the current value of U to Y(Ei), that is, without first copying Y0 to Y. These actions occur in a single time step.
View Sample Time Using the Digital Clock Block

This example shows how to view the simulation sample time at a specified sampling interval using the Digital Clock block. In this model, the Scope shows the output of a Digital Clock block with the Sample time set to 0.2.

In this configuration, the Digital Clock block outputs the simulation time every 0.2 seconds. Otherwise, the block holds the output at the previous value.
Bit Specification Using a Positive Integer

This example shows how to specify the **Number of bits** in the Counter Free-Running block as a positive integer.

At $t = 255$, the counter reaches the maximum value of $(2^8) - 1$. If you increase the stop time of the simulation to 256, the counter wraps to zero.
Bit Specification Using an Unsigned Integer Expression

This example shows how to specify the **Number of bits** in the Counter Free-Running block as an unsigned integer expression.

At $t = 254$, the counter reaches the maximum value of $\text{uint8}(2^{\text{uint8}(8)}-1)$. If you increase the stop time of the simulation to 255, the counter wraps to zero.
Track Running Minimum Value of Chirp Signal

This example shows how to track the running minimum value of a signal generated by the Chirp Signal block.

![Diagram of Chirp Signal block and MinMax Running Resettable block](image)

The Chirp Signal block generates a sine wave whose frequency increases at a linear rate with time. The MinMax Running Resettable block tracks the minimum value of that chirp.
signal over time. The running minimum value is reset every 5 seconds by the Pulse Generator block.
Horizontal Matrix Concatenation

This example shows how to perform a horizontal matrix concatenation with the Matrix Concatenate block. When you set the **Concatenate dimension** parameter to 2 and the inputs are 2-D matrices, the block performs horizontal matrix concatenation and places the input matrices side-by-side to create the output matrix.
Vertical Matrix Concatenation

This example shows how to perform a vertical matrix concatenation with the Matrix Concatenate block. When you set the **Concatenate dimension** parameter to 1 and the inputs are 2-D matrices, the block performs vertical matrix concatenation and places the input matrices on top of each other to create the output matrix.
Multidimensional Matrix Concatenation

This example shows how to perform multidimensional matrix concatenation with the Matrix Concatenate block. When you set the Concatenate dimension parameter to 3 and the inputs are 2-D matrices, the block performs multidimensional matrix concatenation.
Unary Minus of Matrix Input

This example shows how to compute the unary minus of a matrix input.
Sample Time Math Operations Using the Weighted Sample Time Math Block

This example shows how to add the sample time value to a signal using the Weighted Sample Time Math block.

Using the Weighted Sample Time block, you can see the sample time of this model is 0.2. When you set the Operation parameter to + and the Weight value to 1 on the Weighted Sample Time Math block, the block adds the sample time value of 0.2 to the input signal. When you set the Weight value to 3 in the Weighted Sample Time Math1 block, the block adds Ts*3 to the input signal, thus increasing each value by 0.6.
Construct Complex Signal from Real and Imaginary Parts

This example shows how to use the Real-Imag to Complex block to construct a complex-valued signal from real and imaginary parts. You can provide both the real and imaginary parts as block inputs, or provide one value as an input, and the other on the block dialog box.
Construct Complex Signal from Magnitude and Phase Angle

This example shows how to use the Magnitude-Angle to Complex block to construct a complex-valued signal. You can provide both the magnitude and phase angle as block inputs, or provide one value as an input, and the other on the block dialog box.
Find Nonzero Elements in an Array

This example shows how to use the Find block to find nonzero elements in an array. In the following model, the block is configured to output both the one-based linear index and the value of each nonzero element.

```
ans =

    Simulink.SimulationOutput:
        tout: [51x1 double]

    SimulationMetadata: [1x1 Simulink.SimulationMetadata]
        ErrorMessage: [0x0 char]
```
Calculate the Running Minimum Value with the MinMax Running Resettable Block

This example shows how to use the MinMax Running Resettable block to calculate the running minimum value. To watch how the running minimum value changes at each time step, you can use the Step Forward button to advance the simulation one step at a time.

After running the full simulation, you can view the results in the Scope. The initial value of the running minimum is 0. It begins tracking the Sine Wave signal when the sine wave values turn negative. When the MinMax Running Resettable block receives a reset signal at T=8, the block resets the running minimum value to 0. The running minimum value tracks at 0 for a few time steps, until the sine wave values turn negative again.

ans =

Simulink.SimulationOutput:
    tout: [26x1 double]

SimulationMetadata: [1x1 Simulink.SimulationMetadata]
    ErrorMessage: [0x0 char]
Find Maximum Value of Input

This example shows how to use the MinMax block to output the maximum value of two sine waves.

After running the full simulation, you can view the results in the Scope. Initially, the maximum value (orange line) tracks SineWave2. When the SineWave2 values turn negative, the maximum value begins tracking SineWave1. When the SineWave2 values become positive again, the maximum value resumes tracking SineWave2.

ans =

Simulink.SimulationOutput:
    tout: [57x1 double]

SimulationMetadata: [1x1 Simulink.SimulationMetadata]
    ErrorMessage: [0x0 char]
Permute Array Dimensions

This example shows how to use the Permute Dimensions block to permute the first and third dimensions of a 3-by-4-by-5 input array.

When you set the **Order** parameter to [3, 2, 1], the block permutes the first and third dimensions, and outputs a 5-by-4-by-3 array.
Multiply Inputs of Different Dimensions with the Product Block

This example shows how to perform element-wise (.* ) multiplication of inputs using the Product block. In this example, the Product block multiplies two scalars, a scalar and a vector, and two 2x2 matrices.
Multiply and Divide Inputs Using the Product Block

This example shows how to multiply and divide several input signals using the Product block.
Divide Inputs of Different Dimensions Using the Divide Block

This example shows how to perform element-wise (.* ) division of two inputs using the Divide block. In this example, the Divide block divides two scalars, a vector by a scalar, a scalar by a vector, and two matrices.
Complex Division Using the Product of Elements Block

This example shows how to perform element-wise complex division using the Product of Elements block.

The top Product of Elements block collapses the matrix input to a scalar by taking successive inverses of the four elements:

- \( y = \frac{\frac{1/2+i}{3}}{\frac{4-i}{5}} \)

The bottom Product of Elements block collapses the matrix input to a vector by taking successive inverses along the second dimension:

- \( y(1) = \frac{1/2+i}{3} \)
- \( y(2) = \frac{1/4-i}{5} \)
Element-Wise Multiplication and Division Using the Product of Elements Block

This example shows how to use the Product of Elements block to perform element-wise multiplication and division of inputs.
**sin Function with Floating-Point Input**

This example shows how to use the Trigonometric Function block to compute the sine of a floating-point input. The output of the Trigonometric Function block has the same data type as the input because the input data type is floating-point and the **Approximation method** is none.
sincos Function with Fixed-Point Input

This example shows how to use the Trigonometric Function block to compute the CORDIC approximation of sincos for a fixed-point input signal.

The Trigonometric Function block parameters are:

- **Function**: sincos
- **Approximation method**: CORDIC
- **Number of iterations**: 11

When using the CORDIC approximation method, the input to the Trigonometric Function block must be in the range \([-2\pi, 2\pi)\). The output type of the Trigonometric Function block is \texttt{fixdt}(1,13,11) because the input is a fixed-point signal and the **Approximation method** is set to CORDIC. The output fraction length equals the input word length minus two.
Trigonometric Function Block Behavior for Complex Exponential Output

This example compares the complex exponential output for two different configurations of the Trigonometric Function block.

When the **Approximation method** is CORDIC, the input data type can be fixed point, in this case: `fixdt(1,16,2)`. The output data type is `fixdt(1,16,14)` because the output fraction length equals the input word length minus two.

When the **Approximation method** is None, the input data type must be floating point. The output data type is the same as the input data type.
Create Bus Object from Constant Block

This example shows how a single Constant block can replace the six Constant blocks that model `ex_bus_initial_conditions` uses to create a bus object.

```matlab
load_system('ex_constantbus');
out = sim('ex_constantbus');
open_system('ex_constantbus');
```

Verify that the output from the Constant block reflects the values from the `constant_value_struct`.

```matlab
constant_value_struct
constant_value_struct =
    struct with fields:
        A: [1x1 struct]
        B: 5
        C: [1x1 struct]
```

Examine the logged data in the `logsout` variable, focusing on the A1 bus signal. The `constant_value_struct` structure sets the B element to 5.

```matlab
out.logsout.get('A1').Values.B.Data(1)
```

```matlab
ans =
    5
```
Control Algorithm Execution Using Enumerated Signal

This example shows how to use a signal of an enumerated data type to control the execution of a block algorithm. For basic information about using enumerated data types in models, see “Use Enumerated Data in Simulink Models”.

Define Enumerated Type

Copy the enumerated type definition `ex_SwitchCase_MyColors` into a script file in your current folder.

```matlab
classdef ex_SwitchCase_MyColors < Simulink.IntEnumType
tenumeration
    Red(0)
    Yellow(1)
    Blue(2)
    Mauve(3)
end
end
```

Alternatively, you can use the function `Simulink.defineIntEnumType` to define the type.

```matlab
Simulink.defineIntEnumType('ex_SwitchCase_MyColors',... 
    {'Red','Yellow','Blue','Mauve'},[0;1;2;3])
```

Explore Example Model

Open the example model `ex_enum_switch_case`.

```matlab
open_system('ex_enum_switch_case')
```
Open the Enumerated Constant block dialog box. The constant output value is `ex_SwitchCase_MyColors.Blue`.

Open the Switch Case block dialog box. The **Case conditions** box is specified as a cell array containing three of the four possible enumeration members. The block has four outputs corresponding to the three specified enumeration members and a default case.

Open the Switch Case Action Subsystem blocks. The subsystems each contain a Constant block that uses a different constant value.

**Control Execution During Simulation**

Simulate the model. The Display block shows the value 5, which corresponds to the case `ex_SwitchCase_MyColors.Blue`.

In the Enumerated Constant block dialog box, specify **Value** as `ex_SwitchCase_MyColors.Red` and click **Apply**. The Display block shows 19.

Specify **Value** as `ex_SwitchCase_MyColors.Mauve` and click **Apply**. The Display block shows 3, which corresponds to the default case.
Integer and Enumerated Data Type Support in the Ground Block

This example shows how to use the Ground block to ground block input ports that have integer and enumerated data types. In top row of this example, the output of the Constant block determines the data type (int8) of the port to which the Ground block is connected. That port determines the output data type of the Ground block, and the Ground block outputs a signal with zero value, and data type int8.

In the bottom row of this example, the Ground block is connected to a port with an enumerated data type. For enumerated data types, the Ground block outputs the default value of the enumeration. This behavior applies whether or not:

- The enumeration can represent zero
- The default value of the enumeration is zero

If the enumerated type does not have a default value, the Ground block outputs the first enumeration value in the type definition.
Fixed-Point Data Type Support in the Ground Block

This example shows how to use the Ground block to ground block input ports that have fixed-point data types. The top row of this example illustrates the Ground block behavior when the fixed-point data type can represent zero. In that case, the Ground block outputs a signal with zero value, and the same fixed-point data type as the port it is connected to.

In the bottom row of this example, the output of the Constant block determines the data type of the port to which the Ground block is connected (fixdt(0,8,1,1)). Because zero cannot be represented exactly by the data type fixdt(0,8,1,1), the Ground block outputs a nonzero value that is the closest possible value to zero (in this case, 1).
Read 1-D Array and Structure From Workspace

This example shows how to read 1-D signals from the MATLAB workspace. When you open the model, the following code is executed by a PreLoadFcn callback:

```matlab
t = 0.2 * [0:49]';
x = sin(t);
y = 10*sin(t);
wave.time = t;
wave.signals.values = [x,y];
wave.signals.dimensions = 2;
```

In the top row of the model, the From Workspace block reads the array `[t, x, y]` from the MATLAB workspace.

In the bottom row of the model, the From Workspace block reads the same values from the workspace, but this time they are read from a structure named `wave`. 
Read Structure From Workspace Using Model Sample Time

This example shows how to read a structure from the MATLAB workspace using a sample time specified in the From Workspace block. When you open the model, the following code is executed by a PreLoadFcn callback:

```matlab
t = 0.2 * [0:49]';
x = sin(t);
y = 10*sin(t);
wave.time = [];
wave.signals.values = [x,y];
wave.signals.dimensions = 2;
```

The From Workspace block is configured as follows:

- **Sample time**: 0.2
- **Interpolate data**: off
- **Form output after final value by**: Setting to zero

When you run the model, the From Workspace block reads the structure `wave` from the workspace. After the last time hit for which workspace data is available, the block outputs 0.
Read Structure From Workspace Using Model Sample Time
Read 2-D Signals in Structure Format From Workspace

This example shows how to read a 2-D structure from the MATLAB workspace. When you open the model, the following code is executed by a PreLoadFcn callback:

\[
\begin{align*}
\text{t1} &= 0.2 \times [0:49]'; \\
\text{m} &= \text{magic}(10); \\
\text{M} &= \text{repmat}(\text{m},[1 1 \text{length(t1)}]); \\
\text{data.time} &= \text{t1}; \\
\text{data.signals.values} &= \text{M}; \\
\text{data.signals.dimensions} &= [10 \ 10];
\end{align*}
\]

This code creates a 10-by-10 matrix (2-D signal) by using the `magic` function, and then creates a 3-D matrix by adding a time vector. The time vector must be a column vector. The `signals.values` field is a 3-D matrix where the third dimension corresponds to time. The `signals.dimensions` field is a two-element vector. The first element is the number of rows and the second element is the number of columns in the `signals.values` field.

When you run the model, the From Workspace block reads the structure `data` from the workspace.
From File Block Loading Timeseries Data

Create a MATLAB® timeseries object with time and signal values. Save the timeseries object to a MAT-file and load into a model using a From File block.

Create an array with the time and signal data, specifying signal data for 10 time steps.

```matlab
t = .1*(1:10);
d = .2*(1:10);
x = [t;d];
```

Create a MATLAB timeseries object.

```matlab
ts = timeseries(x(2:end,:),x(1,:))
```

```
timeseries
  Common Properties:
    Name: 'unnamed'
    Time: [10x1 double]
    TimeInfo: tsdata.timemetadata
    Data: [1x1x10 double]
    DataInfo: tsdata.datametadata
```

Save the timeseries object in a Version 7.3 MAT-file.

```matlab
save('mySignals','ts','-v7.3')
```

Add a From File block and set the File name parameter of that block to `mySignals.mat`.

Simulate the model. The Scope block reflects the data loaded from the `mySignals.mat` file.
Eliminate Singleton Dimension with the Squeeze Block

This example shows a model using the Squeeze block to eliminate a dimension of size 1.

The Squeeze block converts a multidimensional array from the Constant block of size 3-by-1-by-2 into a 3-by-2 signal.
Difference Between Time- and Sample-Based Pulse Generation

This example shows the difference in behavior of the Pulse Generator block in time-based and sample-based modes.

Consider this model, with two Pulse Generator blocks. One block has the **Pulse type** parameter set to Time based, and the other to Sample based. Both blocks are configured to output a pulse with an amplitude of one that is *on* for five seconds, followed by *off* for five seconds. The simulation time runs from three seconds to a stop time of 18 seconds. Notice the time offset notice in the lower right corner.

```matlab
ans =
    Simulink.SimulationOutput:
        tout: [61x1 double]
    SimulationMetadata: [1x1 Simulink.SimulationMetadata]
    ErrorMessage: [0x0 char]
```
Notice that the time-based Pulse Generator produces an on signal for only two seconds and then switches to off. This is due to the block starting to compute the output from t=0 even though it does not output the simulation until t=3. The sample-based block outputs a pulse of five seconds on followed by five seconds off. In this case, the block output does not depend on simulation time and starts only when the simulation starts.
Specify a Waveform with the Repeating Sequence Block

This example shows how you specify a waveform with the Repeating Sequence block. In this model, the block defines the **Time values** parameter as \([0:0.1:0.5]\) and the **Output values** parameter as \([0\ 15\ 25\ 09\ 13\ 17]\). The stop time of the simulation is 0.7 second.
• Input period is 0.5.
• Output at any time $t$ is the output at time $t = t - 0.5n$, where $n = 0,1,2$, and so on.
• Sequence repeats at $t = 0.5n$.

At $t = 0.5$, the expected output is equal to the output at $t = 0$, which is 0. Therefore, the last value in the **Output values** parameter vector $[0\ 15\ 25\ 09\ 13\ 17]$ does not appear.
Tune Phase Delay on Pulse Generator During Simulation

This example shows how to modify a model so that you can change a phase delay for a Pulse Generation block during simulation. You cannot tune the value of the Phase delay parameter during simulation. As a workaround, add a Constant block and a Variable Time Delay block.

In the Pulse Generator block, set the value of the Phase delay parameter to zero. Use the Constant block to specify the delay time in seconds. To tune the delay time during simulation, change the value stored in the Constant block.
Difference Sine Wave Signal

This example shows how to calculate the difference in a sine wave signal at each time step. The input is a 1-by-2 vector of sine waves, with amplitude 1 and 3. The difference block calculates the difference in each sine wave signal at every time step. The Scope block displays both the original sine waves and the output of the difference block.
Discrete-Time Derivative of Floating-Point Input

This example shows how to use the Discrete Derivative block to compute the discrete-time derivative of a floating-point input signal. The unfiltered discrete-time derivative is compared to a filtered discrete-time derivative that is computed by the Discrete Filter block.
First-Order Sample-and-Hold of a Sine Wave

This example shows how to perform a first-order sample-and-hold of a sine wave signal using the First-Order Hold block.
First-Order Sample-and-Hold of a Sine Wave
Calculate and Display Simulation Step Size using Memory and Clock Blocks

This example shows how to use the Memory and Clock blocks to calculate and display the step size in a simulation. The Sum block subtracts the time at the previous time step, which the Memory block generates, from the current time, which the Clock block generates.

Because \textbf{Inherit sample time} is not selected for the Memory block, the block sample time depends on the type of solver for simulating the model. In this case, the model uses a fixed-step solver. Therefore, the sample time of the Memory block is the solver step size, or 1.

If you replace the Memory block with a Unit Delay block, you get the same results. The Unit Delay block inherits a discrete sample time of 1.
Capture the Velocity of a Bouncing Ball with the Memory Block

The `sldemo_bounce` example shows how to use the Second-Order Integrator and Memory blocks to capture the velocity of a bouncing ball just before it hits the ground.

```
open_system('sldemo_bounce');
```

Because **Inherit sample time** is not selected for the Memory block, the block sample time depends on the type of solver for simulating the model. In this case, the model uses a variable-step (ode23) solver. Therefore, the sample time of the Memory block is continuous but fixed in minor time step: [0, 1]. When you run the model, you get these results.

```
sim('sldemo_bounce');
```
If you replace the Memory block with a Unit Delay block, you get the same results. However, a warning also appears due to the discrete Unit Delay block inheriting a continuous sample time.

For more information, see the model description.
Implement a Finite-State Machine with the Combinatorial Logic and Memory Blocks

The ex_sldemo_clutch example shows how you can use the Memory block with the Combinatorial Logic block to implement a finite-state machine. This construct appears in the Friction Mode Logic/Lockup FSM subsystem:

Building a Clutch Lock-Up Model
An Example of Enabled Subsystems
Because **Inherit sample time** is not selected for the Memory block, the block sample time depends on the type of solver for simulating the model. In this case, the model uses a
variable-step (ode23) solver. Therefore, the sample time of the Memory block is continuous but fixed in minor time step: $[0, 1]$.

For more information, see the model description.
Discrete-Time Integration Using the Forward Euler Integration Method

The sldemo_fuelsys model uses a Discrete-Time Integrator block in the fuel_rate_control/airflow_calc subsystem. This block uses the Forward Euler integration method.

When the Switch block feeds a nonzero value into the Discrete-Time Integrator block, integration occurs. Otherwise, integration does not occur.

For more information, see the model description.
Signal Routing with the From, Goto, and Goto Tag Visibility Blocks

This example shows how to use the From, Goto, and Goto Tag Visibility blocks to route signals in your model. The GotoGlobal block at the top-level of the model has the Goto tag parameter set to G and the Tag visibility set to global. Thus, the G tag can be seen by From and Goto blocks at any level of the model hierarchy, except locations that span nonvirtual subsystem boundaries (like the Atomic Subsystem in this model). The From block at the top level of the model can see and connect to the global G tag, but cannot see or connect to the scoped S tag or L local tag that are specified on Goto blocks further down in the model hierarchy.

Inside of the Main Subsystem, the Goto block with Goto tag set to S has a Tag visibility of scoped. The Goto Tag Visibility block placed at the same level as that Goto block indicates the S tag can be seen by all From and Goto blocks at that level and below, except for locations that cross a nonvirtual subsystem boundary (i.e. the boundary with the Atomic Subsystem). Inside of Subsystem1 and Subsystem2, the From blocks can see and connect to the global Goto tag G, and the scoped Goto tag S.
Signal Routing with the From, Goto, and Goto Tag Visibility Blocks
Zero-Based and One-Based Indexing with the Index Vector Block

This example shows how the Index Vector block works with zero-based and one-based indexing.

The Index Vector block is from the Simulink Signal Routing library. It is a special configuration of the Multiport Switch block. To configure the Multiport Switch block as an Index Vector block, set the Number of data ports to 1 and Data port order to Zero-based contiguous.
Noncontiguous Values for Data Port Indices of Multiport Switch Block

This example shows how to use a Multiport Switch block that specifies noncontiguous integer values for data ports. The values of the indices are visible on the data port labels. You do not have to open the block dialog box to determine which value maps to each data port.

When you set Data port for default case to Additional data port, an extra port with a * label appears. This port corresponds to the default case, which applies when the control input does not match the data port indices 3, 5, 0, or 18. When that happens in this example, the Multiport Switch block outputs a value of 1.
Using Variable-Size Signals on the Delay Block

This example shows how the Delay block supports variable-size signals for sample-based processing. The Switch block controls whether the input signal to the enabled subsystem is a 3-by-3 or 3-by-2 matrix.

The Delay block appears inside the enabled subsystem.

The model follows these rules for variable-size signals while using sample-based processing.

- Signal dimensions change only during state reset when the block is enabled.
• Initial condition must be scalar.

The rules are implemented by these blocks.

• Enable block sets Propagate sizes of variable-size signals to Only when enabling.
• Delay block sets the Initial condition to the scalar value 0.0.
Bus Signals with the Delay Block for Frame-Based Processing

This example shows how the Delay block supports bus signals for frame-based processing.

Each Constant block supplies an input signal to the Bus Creator block, which outputs a two-dimensional bus signal. After the Delay block delays the bus signal by three sample periods, the Bus Selector block separates the bus back into the two original signals.

The model follows these rules for bus signals.

- For the initial condition, set the value on the dialog box.
- For frame-based processing, signal dimensions of the data input port \( u \) cannot be larger than two.

The model implements the rules by:

- Setting the **Initial condition** to a scalar value of 0.
- Setting bus input to the Delay block as two dimensions.
Control Execution of Delay Block with Enable Port

This example shows how you can enable or disable the execution of the Delay block using an enable port. In this model, a ramp input signal feeds into a Delay block. The execution of the block is controlled by an enabling signal from the Pulse Generator block.

The blue line shows that the Delay block outputs the input signal value delayed by one time step while the enabling signal has a value of one. At $t=5$ the enabling signal
transitions to zero and the Delay block stops executing. The output is held at the last value until the block is enabled again.
Zero-Based Indexing for Multiport Switch Data Ports

The sf_aircontrol model uses a Multiport Switch block in the Physical Plant subsystem. This block uses zero-based indexing for contiguous ordering of three data ports.

The indices are visible on the data port labels. You do not have to open the block dialog box to determine if the data ports use zero-based or one-based indexing.

When you set Data port for default case to Last data port, the last data port includes a * on the label (in this case, the label is *, 2). The comma after the * indicates that the data port index has a value. This port corresponds to the default case, which applies when the control input does not match the data port indices. In this example, the Multiport Switch block outputs a value of -0.1 when the control input does not match the data port indices of 0, 1, or 2.
One-Based Indexing for Multiport Switch Data Ports

The `sf_semantics_hotel_checkin` model uses a Multiport Switch block. This block uses one-based indexing for contiguous ordering of three data ports.
When you increase the size of the block icon, the indices are visible on the data port labels. You do not have to open the block dialog box to determine whether the data ports use zero-based or one-based indexing.
Enumerated Names for Data Port Indices of the Multiport Switch Block

The sldemo_fuelsys model uses a Multiport Switch block in the fuel_rate_control/fuel_calc/feedforward_fuel_rate subsystem. This block uses the enumerated type sld_FuelModes to specify three data port indices: LOW, RICH, and DISABLED.

When you set **Data port for default case** to Last data port, the last data port includes a * on the label. The comma and ellipsis after the * indicate that the data port index has a value. This port corresponds to the default case, which applies when the control input does not match the data port indices LOW, RICH, or DISABLED. In this case, the Multiport Switch block outputs a value of 0.
Prevent Block Windup in Multiloop Control

This example shows how to use signal tracking to prevent block windup in a two-loop control system.

The Inner Loop subsystem contains a saturation limit:

In this example, the inner loop has an effective gain of 1 when it does not saturate. When the inner loop does saturate, however, the integrator in the PID Controller can begin to wind up.

If the PID controller tracks the output of the inner loop, then its output never exceeds the saturated inner-loop output. To achieve this tracking, connect the Saturation block output to the tracking input of the PID Controller.
**Bumpless Control Transfer**

This example shows how to use signal tracking to achieve bumpless control transfer in a system that switches between a PID Controller block and another controller. You can make the PID controller track the output of the other controller by connecting the signal you want to track to the TR port of the PID Controller block. For example:

![Block diagram showing the connection](image)

The input In1 takes a reference signal that feeds both controllers. The output Out1 drives a controlled system (not shown). A switch transfers control between the Active controller block (a Zero-Pole block) and the PID Controller block. When the Active controller is in the loop, the PID Controller block tracks its output. Thus, the two controllers have the same output when you switch to PID control, ensuring smooth operation.
Bumpless Control Transfer with a Two-Degree-of-Freedom PID Controller

This example shows how to use signal tracking to achieve bumpless control transfer in a system that switches between a PID Controller (2DOF) block and another controller. You can make the PID controller track the output of the other controller by connecting the signal you want to track to the TR port of the PID Controller block. For example:

The input Ref takes a reference signal, and the input y takes the measured feedback from the plant. These signals feed the 2DOF controller, and the difference between them feeds the Active controller block (a Zero-Pole block). The output u drives the plant. A switch transfers control between the active controller and the PID Controller (2DOF) block. When the Active controller is in the loop, the PID Controller (2DOF) block tracks its output. Thus, the two controllers have the same output when you switch to PID control, ensuring smooth operation.
Using a Bit Set block

If the Bit Set block is turned on for bit 2 bit 2 is set to 1.

A vector of constants \(2.^[0 1 2 3 4]\) is represented in binary as \([00001 00010 00100 01000 10000]\).

With bit 2 set to 1, the result is \([00101 00110 00100 01100 10100]\), which is represented in decimal as \([5 6 4 12 20]\).
Using a Bit Clear block

If the Bit Clear block is turned on for bit 2, bit 2 is set to 0.

A vector of constants $2.^[0 1 2 3 4]$ is represented in binary as $[00001 00010 00100 01000 10000]$.

With bit 2 set to 1, the result is $[00101 00110 00100 01100 10100]$, which is represented in decimal as $[5 6 4 12 20]$.

With bit 2 set to 0, the result is $[00001 00010 00000 01000 10000]$, which is represented in decimal as $[1 2 0 8 16]$. 
Two-Input AND Logic

This example builds a two-input AND function, which returns 1 when both input elements are 1, and 0 otherwise. To implement this function, specify the Truth table parameter value as [0; 0; 0; 1] The portion of the model that provides the inputs to and the output from the Combinatorial Logic block might look like this:

The following table indicates the combination of inputs that generate each output. The input signal labeled Input corresponds to the column in the table labeled Input 1. Similarly, the input signal Input 2 corresponds to the column with the same name. The combination of these values determines the row of the Output column of the table that is passed as block output. For example, if the input vector is [1 0], the input references the third row:

\[(2^1*1 + 1)\] The output value is 0.

<table>
<thead>
<tr>
<th>Row</th>
<th>Input 1</th>
<th>Input 2</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Circuit Logic

This sample circuit has three inputs: the two bits (a and b) to be summed and a carry-in bit (c). It has two outputs: the carry-out bit (c') and the sum bit (s).

The truth table and corresponding outputs for each combination of input values for this circuit appear in the following table.

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

To implement this adder with the Combinatorial Logic block, you enter the 8-by-2 matrix formed by columns c' and s as the Truth table parameter. You can also implement sequential circuits (that is, circuits with states) with the Combinatorial Logic block by including an additional input for the state of the block and feeding the output of the block back into this state input.
Unsigned Inputs for the Bitwise Operator Block

The following model shows how the Bitwise Operator block works for unsigned inputs.

Each Constant block outputs an 8-bit unsigned integer (uint8). To determine the binary value of each Constant block output, use the dec2bin function. The results for all logic operations appear in the next table.
Signed Inputs for the Bitwise Operator Block

The following model shows how the Bitwise Operator block works for signed inputs.

Each Constant block outputs an 8-bit signed integer (int8). To determine the binary value of each Constant block output, use the dec2bin function. The results for all logic operations appear in the next table.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Binary Value</th>
<th>Decimal Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND</td>
<td>01000000</td>
<td>64</td>
</tr>
<tr>
<td>OR</td>
<td>11110111</td>
<td>-5</td>
</tr>
<tr>
<td>NAND</td>
<td>10111111</td>
<td>-55</td>
</tr>
<tr>
<td>NOR</td>
<td>00000100</td>
<td>4</td>
</tr>
<tr>
<td>XOR</td>
<td>11000101</td>
<td>-52</td>
</tr>
<tr>
<td>NOT</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>
Merge Block with Input from Atomic Subsystems

This example shows a Merge block with inputs from two atomic subsystems.

Each Atomic Subsystem block contains an enabled subsystem. This satisfies the requirement that inputs to a Merge block are from a conditionally executed subsystem.
Index Options with the Selector Block

This example shows two Selector blocks with the same kind of input signals, but two different Index Option settings.
Both Selector blocks select 7 values from the input signal that feeds the input port. The Selector1 block outputs a fixed-size signal, whereas the Selector2 block outputs a variable-size signal whose compiled signal dimension is 10 instead of 7.

The Selector1 block sets Index Option to Index vector (port), which uses the input signal from Constant1 as the index vector. The dimension of the input signal is 7, so the Display block shows the 7 values of the Constant1 block. The Selector2 block sets the Input port size parameter to 10, which is the size of the largest input signal to the Selector2 block.

The Selector2 block also sets the Index Option to Starting and ending indices (port). The output is then set to the size of Input port size parameter (10), even though the size of the input signal is 7.
Switch Block with a Boolean Control Port Example

This example shows a Switch block with a Boolean input for the control port.

open_system('sldemo_fuelsys');
open_system('sldemo_fuelsys/fuel_rate_control/airflow_calc');

The value of the control port on the Switch block determines whether or not the feedback correction occurs. The control port value depends on the output of the Logical Operator block. When the Logical Operator block output is true, then the Switch block control port is 1 and the feedback control occurs. If the Logical Operator block output is false then the feedback control does not occur.
Merge Block with Unequal Input Widths Example

This example shows how to use the Merge block with inputs ports that have different widths. If you select Allow unequal port widths, the block accepts scalar and vector inputs having differing numbers of elements. You can specify an offset for each input signal relative to the beginning of the output signal. The width of the output signal is:

\[ \text{max}(w_1 + o_1, w_2 + o_2, \ldots w_n + o_n) \]

where \( w_n \) are the widths of the input signals, and \( o_n \) are the offsets.
The Merge block has the following output width.

$$\max(2 + 0, 2 + 1) = 3$$

In this example, the offset of $v_1$ is 0 and the offset of $v_2$ is 1. The Merge block maps the elements of $v_1$ to the first two elements of $v_3$ and the elements of $v_2$ to the last two.
elements of $v_3$. Only the second element of $v_3$ is effectively merged, as show in the scope output.

If you use Simplified Initialization Mode, you must clear the Allow unequal port widths check box. The input port offsets for all signals must be zero.
Detect Rising Edge of Signals

This example shows how to detect the rising edge of a signal using the Detect Rise Nonnegative and Detect Rise Positive blocks.
With a fixed-step size of 0.25, this example illustrates the difference between the Detect Rise Nonnegative and Detect Rise Positive blocks. The Detect Rise Nonnegative block outputs true (1) at \( t=1 \) because the input signal increased from a negative value to a nonnegative value (0). The Detect Rise Positive block outputs true (1) at \( t=1.25 \) because the input signal increased from a nonpositive value (0) to a strictly positive value.
Detect Falling Edge Using the Detect Fall Nonpositive Block

This example shows how to use the Detect Fall Nonpositive block to detect a falling edge in the input signal. The block detects a falling edge when the signal value decreases from a strictly positive value to a nonpositive value. In this example, the Initial condition of the Detect Fall Nonpositive block is set to 1. This means that the Boolean expression $U/z \leq 0$ evaluates to true and the block assumes the initial value of the input signal is nonpositive.
Detect Increasing Signal Values with the Detect Increase Block

This example shows how to use the Detect Increase Block to detect increasing signal values. Because the **Initial condition** is set to -1, the block detects an increasing signal value starting at time $t=0$. If you change the **Initial condition** parameter to a nonnegative value, the block detects the first increasing signal value at $t=0.25$. 

![Diagram of Detect Increase Block](image-url)
Extract Bits from Stored Integer Value

This example shows how to extract specific bits from the stored integer value of an input signal.
Detect Signal Values Within a Dynamically Specified Interval

This example shows how to detect when an input signal falls within a dynamically specified interval. The interval is defined by two Sine Wave blocks. When the input to the Interval Test Dynamic block falls between those sine waves, the Interval Test Dynamic block outputs true (1).
Detect Signal Values Within a Dynamically Specified Interval
Model a Digital Thermometer Using the Polynomial Block

This example shows how the `ex_sldemo_boiler` model uses the Polynomial block.

In the Boiler Plant model/digital thermometer subsystem, the Polynomial Block models a first-order polynomial using the coefficients [0.05 0.75]
Convert Data Types in Simulink Models

This example shows three different methods of converting data types in your model using the Data Type Conversion and Data Type Conversion Inherited blocks. In this model, a Sine Wave block generates the input signal. The Sine Wave block only outputs double-precision data types, so to generate a sine wave with a data type of single, you must perform a data type conversion.
In the first row, the Data Type Conversion Inherited block uses the data type coming from the Constant block (single) as the reference data type, and converts the sine wave to single.

In the second row, the Data Type Conversion block has the **Output data type** set to single, and the sine wave is converted accordingly.

In the third row, the Data Type Conversion1 block has the **Output data type** set to **Inherit: Inherit via back propagation**. Because the downstream Gain2 block has a data type of single, the Data Type Conversion1 block converts the sine wave to a data type of single.
Control Data Types with the Data Type Duplicate Block

This example shows how to control data types in your model using the Data Type Duplicate block. In this model, the data type of the Constant block (currently `single`) drives the data types throughout the model.

The Constant1 block has its **Output data type** parameter set to Inherit: Inherit via back propagation. Because the Constant1 and Constant blocks are both connected to the Data Type Duplicate block, the Constant1 block can inherit its data type from the Constant block. The Sum block has its **Output data type** set to Inherit: Same as first input, so it is also able to inherit its data type from the Constant block.

If you change the data type of the Constant block from `single` to `int32`, the `int32` data type propagates throughout the model.
Probe Sample Time of a Signal

The sldemo_fuelsys model shows how to check the sample time of a signal using the Probe block. This enables you to verify that the sample time matches the assumed value of the design.

Verify sample time: aspects of the design assume a 0.01 second sample time.

The contents of this subsystem are excluded from the code Simulink Coder generates.

To work correctly this system must continue to
  * Contain no output ports
  * Enable its 'Treat as Atomic Unit' parameter
  * Specify its 'Mask type' parameter as "VerificationSubsystem"

For more information, see the model description.
The sldemo_fuelsys model shows how to use the Rate Transition block to convert signals between continuous time and discrete time.

In the To Controller subsystem, the Rate Transition block converts the signal from continuous time to discrete time. This discrete-time signal can then be processed by the fuel_rate_control subsystem.
The To Plant subsystem uses the Rate Transition block to convert the discrete-time output of the fuel_rate_control subsystem back to continuous time.

For more information, see the model description.
Convert Muxed Signal to a Vector

This example shows how to use the Signal Conversion block to convert a muxed input signal to a vector.

To convert the muxed signal to a vector, the **Output** parameter of the Signal Conversion block is set to **Signal copy**.
Create Contiguous Copy of a Bus Signal

This example shows how to use the Signal Conversion block to create a contiguous copy of a bus signal.

The Bus Creator block creates a nonvirtual bus signal that is input to the Signal Conversion block. With the **Output** parameter set to **Signal copy**, the Signal Conversion block creates another contiguous copy of that input bus signal.
Convert Virtual Bus to a Nonvirtual Bus

In the following example, the Signal Conversion block converts a virtual bus signal from the first Bus Selector block to a nonvirtual bus signal that inputs to the second Bus Selector block. The Signal Conversion block has its **Output** parameter set to **Nonvirtual bus**, and specifies a bus object that matches the bus signal hierarchy of the bus that the first Bus Creator block outputs.
Convert Nonvirtual Bus to Virtual Bus

In the following example, the Signal Conversion block converts the nonvirtual bus signal from the Bus Creator block to a virtual bus signal that inputs to the Bus Selector block. The Signal Conversion block has its **Output** set to **Virtual bus**.
Remove Scaling from a Fixed-Point Signal

This example shows how to strip the scaling from a fixed-point input signal. To do so, the Data Type Scaling Strip block maps the input to the smallest built-in data type that has enough data bits to hold the input. The stored integer value of the input is the value of the output.
Stop Simulation Block with Relational Operator Block

This example shows how to control when a simulation stops by using a Stop Simulation block with a Relational Operator block. When you simulate the model, the model stops simulation when the simulation time reaches 10.
Output Simulation Data with Blocks

This example shows how To Workspace and To File blocks write data to the workspace and to a file respectively.

Open Example Model

open_system('ex_ToWorkspace_ToFile');

Simulate with Default Parameter Values

1. To name the output variables and file, modify the Variable name and File name block parameter values by using the Block Parameters dialog boxes or the command line.

set_param('ex_ToWorkspace_ToFile/To Workspace','VariableName','simoutToWorkspace')

set_param('ex_ToWorkspace_ToFile/To File','FileName','simoutToFile.mat','MatrixName','simoutToFileVariable')

2. Simulate the model.

out = sim('ex_ToWorkspace_ToFile');

3. To view the input signal for the To Workspace and To File blocks, open the scope viewer.

4. To access the data stored by the To File block, load the output file.
load('simoutToFile.mat')

5. Plot the data stored by the To Workspace and To File blocks.

```matlab
subplot(2,1,1)
plot(out.simoutToWorkspace,'-o')
legend('simoutToWorkspace')

subplot(2,1,2)
plot(simoutToFileVariable,'-o')
legend('simoutToFileVariable')
```

As shown by the plots, the data stored by each block is the same given the default block parameter values.
Simulate with Custom Parameter Values

1. To keep the data from the previous simulation, specify new names for the output variables and file.

```matlab
set_param('ex_ToWorkspace_ToFile/To Workspace',...  
'VariableName','simoutToWorkspace2')

set_param('ex_ToWorkspace_ToFile/To File',...  
'FileName','simoutToFile2.mat',...  
'MatrixName','simoutToFileVariable2')
```

2. To change the amount of data collected, modify the Limit data points to last, Decimation, and Sample time block parameter values.

```matlab
set_param('ex_ToWorkspace_ToFile/To Workspace',...  
'MaxDataPoints','3',...  
'Decimation','20',...  
'SampleTime','0.5')

set_param('ex_ToWorkspace_ToFile/To File',...  
'Decimation','20',...  
'SampleTime','1')
```

The To File block does not provide the option to limit data points to the last data points collected.

3. Select Single simulation output, then modify the Logging intervals configuration parameter value.

```matlab
set_param('ex_ToWorkspace_ToFile',...  
'ReturnWorkspaceOutputs','on',...  
'LoggingIntervals','[20,90]')
```

4. Simulate the model.

```matlab
out = sim('ex_ToWorkspace_ToFile');
```

5. To access the data stored by the To File block, load the output file.

```matlab
load('simoutToFile2.mat')
```

6. Plot the data stored by the To Workspace and To File blocks.

```matlab
subplot(2,1,1)  
hold on
```

```matlab
14-167```
In this example, the To Workspace block collects data at 20, 30, 40, ..., 90 seconds. The data represents every 20th sample time within the logging intervals. When the simulation is completed or paused, the To Workspace block writes only the last three collected sample points to the workspace: 70, 80, and 90 seconds.
The To File block collects data at 20, 40, 60, and 80 seconds. The data similarly represents every 20th sample time within the logging intervals. However, the sample time for the To File block is double the sample time for the To Workspace block.
Increment and Decrement Real-World Values

This example shows how to increase and decrease the real-world value of a signal using the following blocks:

- Increment Real World
- Decrement Real World
- Decrement Time To Zero
- Decrement To Zero

The Scope block shows the output of a Sine Wave block with amplitude 5, as well as the real-world value of that signal incremented and decremented by one.
The Scope1 block shows the output of a Sine Wave block with amplitude 3, as well as the output of the Decrement To Zero and Decrement Time To Zero blocks:

- The Decrement To Zero block decreases the input sine wave signal by one, and ensures the value never goes below zero.
- The Decrement Time To Zero block decreases the input sine wave signal by the sample time, $T_s$, and ensures that the value never goes below zero.
Increment and Decrement Stored Integer Values

This example shows how to increase and decrease the stored integer value of a signal by one.

- The Increment Stored Integer block increases the stored integer value of the input signal by one.
- The Decrement Stored Integer block decreases the stored integer value of the input signal by one.

If you change the value of the input signal to 127 (the maximum value representable by an int8 data type), incrementing the stored integer value by one causes an overflow. Because overflows in the Increment and Decrement Stored Integer blocks always wrap, the Increment Stored Integer block will output a value of -128.
Specify a Vector of Initial Conditions for a Discrete Filter Block

This example shows how to specify a vector of non-zero initial conditions for the Discrete Filter block.
The Scope shows that with the **Initial states** of the Discrete Filter block set to \([1 \ 2]\), the difference between the signal filtered by the Discrete Filter block and the signal from the filter's building blocks is zero. This demonstrates that you can enter the initial conditions of the Discrete Filter block as a vector of \([1 \ 2]\). As an alternative, you can set the initial condition of the Unit Delay Block to 2. The resulting output is the same.
Generate Linear Models for a Rising Edge Trigger Signal

You can use state and simulation time logging to extract the model states at operating points. In this example, the model is configured to get the states when the \( x_1 \) signal triggers the Trigger-Based Linearization block on a rising edge.

In this model, the **Trigger type** of the Trigger-Based Linearization block is set to **rising**. On the Data Import/Export pane of the Model Configuration Parameters dialog box, the **States** and **Time** check boxes are selected.

![Diagram](image-url)
After simulating the model, the following variables appear in the MATLAB workspace:

- `ex_vdp_triggered_linearization_Trigger_Based_Linearization`
- `tout`
- `xout`

To get the index to the first operating point time, execute the following command:

```matlab
ind1 = find(ex_vdp_triggered_linearization_Trigger_Based_Linearization(1).OperPoint.t==tout);
```

To get the state vector at this operating point, execute the following command:

```matlab
x1 = xout(ind1,:);
```
Generate Linear Models at Predetermined Times

This example shows how to use the Timed-Based Linearization block to generate linear models at predetermined times.

In this model, the Linearization time of the Timed-Based Linearization block is set to [2 5]. On the Data Import/Export pane of the Model Configuration Parameters dialog box, the States and Time check boxes are selected. These settings enable you to get the states of the model at the simulation times of 2 and 5 seconds.

After simulating the model, the following variables appear in the MATLAB workspace:

- `ex_f14_linearization_Timed_Based_Linearization`
• tout
• xout

To get the indices to the operating point times, execute the following command:

```matlab
ind1 = find(ex_f14_linearization_Timed_Based_Linearization(1).OperPoint.t==tout);
```

To get the state vectors at the operating points, execute the following command:

```matlab
ind2 = find(ex_f14_linearization_Timed_Based_Linearization(1).OperPoint.t==tout);
```
Capture Measurement Descriptions in a DocBlock

This example shows how to use a DocBlock to capture information about a model. In the sldemo_fuelsys model, a DocBlock labeled Sensor Info is used to document measurement descriptions inside of the To Controller subsystem.

For more information, see the model description.
Square Root of Negative Values

This example shows how to compute the square root of a negative-valued input signal as complex-valued output.

ans =

Simulink.SimulationOutput:
  tout: [51x1 double]

SimulationMetadata: [1x1 Simulink.SimulationMetadata]
ErrorMessage: [0x0 char]

By setting the **Function** to `sqrt` and **Output signal type** to `complex`, the block produces the correct result of $0 + 10i$ for an input of `-100`. If you change the **Output signal type** to auto or real, the block outputs NaN.
Signed Square Root of Negative Values

This example shows how to compute the signed square root of a negative-valued input signal.

ans =

Simulink.SimulationOutput:
    tout: [51x1 double]

SimulationMetadata: [1x1 Simulink.SimulationMetadata]
ErrorMessage: [0x0 char]

When the block input is negative and you set the Function to signedSqrt, the Sqrt block output is the same for any setting of the Output signal type parameter. By setting the Format of the first Display block to decimal (Stored Integer), you can see the value of the imaginary part for the complex output.
rSqrt of Floating-Point Inputs

This example shows how to compute the rSqrt of a floating-point input signal. The Sqrt block has the following settings:

- **Method** = Newton-Raphson
- **Number of iterations** = 1
- **Intermediate results data type** = Inherit: Inherit from input

After one iteration of the Newton-Raphson algorithm, the block output is within 0.0004 of the final value (0.4834).

```
ans =

Simulink.SimulationOutput:
    tout: [51x1 double]

SimulationMetadata: [1x1 Simulink.SimulationMetadata]
    ErrorMessage: [0x0 char]
```
rSqrt of Fixed-Point Inputs

This example shows how to compute the rSqrt of a fixed-point input signal. The Sqrt block has the following settings:

- **Method** = Newton-Raphson
- **Number of iterations** = 1
- **Intermediate results data type** = Inherit: Inherit from input

After one iteration of the Newton-Raphson algorithm, the block output is within 0.0459 of the final value (0.4834).

```
ans =
Simulink.SimulationOutput:
   tout: [51x1 double]
SimulationMetadata: [1x1 Simulink.SimulationMetadata]
   ErrorMessage: [0x0 char]
```
Model a Series RLC Circuit

Physical systems can be described as a series of differential equations in an implicit form, $F\left(t, x, \{\dot{x}\}\right) = 0$, or in the implicit state-space form $E\dot{x} = A\, x + B\, u$.

If $E$ is nonsingular, then the system can be easily converted to a system of ordinary differential equations (ODEs) and solved as such:

$$\dot{x} = (E^{-1}A)\, x + (E^{-1}B)\, u$$

Many times, states of a system appear without a direct relation to their derivatives, usually representing physical conservation laws. For example:

$$\begin{align*}
\dot{x}_1 &= x_2 \\
0 &= x_1 + x_2
\end{align*}$$

In this case, $E$ is singular and cannot be inverted. This class of systems are commonly called descriptor systems and the equations are called differential-algebraic equations (DAEs).

**Series RLC Circuit**

Consider the simple series RLC circuit.

![RLC Circuit Diagram]

From Kirchoff’s Voltage Law, the voltage drop across the circuit is equal to the sum of the voltage drop across each of its elements:
\[ V_{AC} = V_R + V_L + V_C \]

From Kirchoff’s Current Law:
\[ I_{AC} = I_R = I_L = I_C \]

where the subscripts \( R, L, \) and \( C \) denote the resistance, inductance, and capacitance respectively.
\[ V_R = I(t) R \]

\[ V_L = L\dot{I}_L \text{ or } \dot{I}_L = \frac{1}{L} V_L \]

\[ V_C = V_{AC}(0) + \int_0^t I_C(\tau) \, d\tau \text{ or } \dot{V}_C = \frac{1}{C} I_c \]

**In Implicit State-Space Form**

Model the system in Simulink with \( R = 10 \, \Omega, \, L = 1 \times 10^{-6} \, H, \, C = 1 \times 10^{-4} F \) to find the voltage across the resistor \( V_R \). To use the Descriptor State-Space block, the system can be written in the implicit, or descriptor, state-space form \( E\dot{x} = Ax + Bu \) as shown below.

\[
\begin{bmatrix}
1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 \\
\end{bmatrix}
\begin{bmatrix}
\dot{V}_C \\
\dot{V}_L \\
\dot{V}_R \\
\dot{I}_L \\
\dot{I}_{AC} \\
\end{bmatrix}
= \begin{bmatrix}
0 & 0 & 0 & \frac{1}{C} & 0 \\
1 & 1 & 1 & 0 & 0 \\
0 & 0 & -1 & R & 0 \\
0 & \frac{1}{L} & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & -1 \\
\end{bmatrix}
\begin{bmatrix}
V_C \\
V_L \\
V_R \\
I_L \\
I_{AC} \\
\end{bmatrix}
+ \begin{bmatrix}
0 \\
-1 \\
0 \\
0 \\
0 \\
\end{bmatrix}
V_{AC}
\]

where \( x = [V_C \, V_L \, V_R \, I_L \, I_{AC}]^T \) is the state vector.

Set \( C = [0 \, 0 \, 1 \, 0 \, 0] \) since the voltage across the resistor is being measured.

Compare this to modeling the system with an algebraic loop in order to find \( V_R \).
The simulation of both models produces identical results. However, the Descriptor State-Space block allows you to make a simpler block diagram and avoid algebraic loops.

**See Also**

“Algebraic Loop Concepts”

“Model Differential Algebraic Equations”
Extract Vector Elements and Distribute Evenly Across Outputs

This example shows how you can use the Demux block to distribute an input signal evenly over the desired number of outputs. For an input vector of length 6, when you set the **Number of outputs** parameter to 3, the Demux block creates three output signals, each of size 2.

```matlab
ans =

    Simulink.SimulationOutput:
        tout: [1x1 double]

    SimulationMetadata: [1x1 Simulink.SimulationMetadata]
    ErrorMessage: [0x0 char]
```

See Also
Demux
Extract Vector Elements Using the Demux Block

When using the Demux block to extract and output elements from a vector input, you can use -1 in a vector expression to indicate that the block dynamically sizes the corresponding port. When a vector expression comprises both positive values and -1 values, the block assigns as many elements as needed to the ports with positive values. The block distributes the remaining elements as evenly as possible over the ports with -1 values.

In this example, the **Number of outputs** parameter of the Demux block is set to [-1, 3, -1]. Thus, the block outputs three signals where the second signal always has three elements. The sizes of the first and third signals depend on the size of the input signal. For an input vector with seven elements, the Demux block outputs two elements on the first port, three elements on the second port, and two elements on the third port.

ans =

```
Simulink.SimulationOutput:
  tout: [11x1 double]

SimulationMetadata: [1x1 Simulink.SimulationMetadata]
  ErrorMessage: [0x0 char]
```
See Also
Demux
Detect Change in Signal Values

This example shows how to detect a change in signal values using the Detect Change block. When the input from the Pulse Generator block remains the same, the Detect Change block outputs zero (false), indicating that there was no change in signal values. When the value of the Pulse Generator block changes, the Detect Change block outputs one (true) indicating that the current signal value does not equal its previous value.
See Also
Detect Change | Pulse Generator | Scope
Detect Fall to Negative Signal Values

This example shows how to detect when a signal value decreases to a strictly negative value from a value that was greater than or equal to zero.
See Also
Detect Fall Negative | Scope | Sine Wave
Detect Decreasing Signal Values

This example shows how to determine if an input signal is strictly less than its previous value using the Detect Decrease block. When the current input to the Detect Decrease block is less than its previous value, the block outputs one (true). When the current input is greater than or equal to the previous signal value, the Detect Decrease block outputs zero (false).
See Also
Detect Decrease | Scope | Sine Wave
Function-Call Blocks Connected to Branches of the Same Function-Call Signal

In this model, a Function-Call Feedback Latch block is on the feedback signal between the branched blocks. As a result, the latch block delays the signal at the input of the destination function-call block, and the destination function-call block executes prior to the source function-call block of the latch block.

See Also
Function-Call Feedback Latch | Function-Call Generator | Function-Call Split | Function-Call Subsystem
Function-Call Feedback Latch on Feedback Signal Between Child and Parent

In this model, the Function-Call Feedback Latch block is on the feedback signal between the child and the parent. This arrangement prevents the signal value, read by the parent (FCSS1), from changing during execution of the child. In other words, the parent reads the value from the previous execution of the child (FCSS2).

See Also
Digital Clock | Function-Call Feedback Latch | Function-Call Generator | Function-Call Subsystem
Single Function-Call Subsystem

In this example, a single function-call subsystem output serves as its own input.

See Also
Digital Clock | Function-Call Feedback Latch | Function-Call Generator | Function-Call Subsystem
**Function-Call Subsystem with Merged Signal As Input**

In this model a merged signal serves as the input to a function-call subsystem. Latching is necessary if one of the signals entering the Merge block forms a feedback loop with the function-call subsystem. In this example, one of the output signals from FCSS2 combines with the output of an Enabled Subsystem block and then feeds back into an inport of FCSS2.

See Also

Digital Clock | Enabled Subsystem | Function-Call Feedback Latch | Function-Call Generator | Function-Call Subsystem | Merge | Pulse Generator
Partitioning an Input Signal with the For Each Block

The following model demonstrates the partitioning of an input signal by a For Each block. Each row of this 2-by-3 input array contains three integers that represent the \((x, y, z)\)-coordinates of a point. The goal is to translate each of these points based on a new origin at \((-20, -10, -5)\) and to display the results.

By placing the process of summing an input signal and the new origin inside of a For Each Subsystem block, you can operate on each set of coordinates by partitioning the input signal into two row vectors. To accomplish such partitioning, use the default settings of 1 for both the partition dimension and the partition width. If you also use the default concatenation dimension of 1, each new set of coordinates stacks in the \(d1\) direction, making your display a 2-by-3 array.

See Also
For Each | For Each Subsystem
Specifying the Concatenation Dimension in the For Each Block

This example shows how to specify a concatenation dimension in the For Each block. When you specify a **Concatenation Dimension** of 2 on the **Output Concatenation** tab, each set of results stacks in the d2 direction, and the result is a single row vector.

To learn how the For Each block and subsystem handle a model with states, see the For Each Subsystem block reference page.

**See Also**
For Each | For Each Subsystem
Working with the Initialize Function, Reset Function, and Terminate Function Blocks

This example shows how to use the Initialize Function, Reset Function, and Terminate Function blocks to read and write states in a Simulink model. For more information on configuring the block settings, see “Using Initialize, Reset, and Terminate Functions”.

See Also
Discrete-Time Integrator | Event Listener | Initialize Function | Reset Function | State Reader | State Writer | Terminate Function

Related Examples
• “Using Initialize, Reset, and Terminate Functions”
Reading and Writing States with the Initialize Function and Terminate Function Blocks

In this example, the Initialize Function block uses the State Writer block to set the initial condition of a Discrete Integrator block to 10. The Terminate Function block includes a State Reader block, which reads the state of the Discrete Integrator block. The Event type parameter of the Event Listener block for the initialize and terminate functions is set to Initialize and Terminate, respectively.

See Also
Discrete-Time Integrator | Event Listener | Initialize Function | State Reader | State Writer | Terminate Function

Related Examples
• “Using Initialize, Reset, and Terminate Functions”
Simulink Featured Examples

- “Simulation of a Bouncing Ball” on page 15-7
- “Single Hydraulic Cylinder Simulation” on page 15-17
- “Thermal Model of a House” on page 15-31
- “Approximating Nonlinear Relationships: Type S Thermocouple” on page 15-38
- “Digital Waveform Generation: Approximating a Sine Wave” on page 15-48
- “Accurate Zero-Crossing Detection” on page 15-61
- “Spiral Galaxy Formation Simulation Using MATLAB Function Blocks” on page 15-63
- “Modulo-4 Counter Using Flip-Flops” on page 15-68
- “Counters Using Conditionally Executed Subsystems” on page 15-70
- “Friction Model with Hard Stops” on page 15-73
- “State Events” on page 15-75
- “Bang-Bang Control Using Temporal Logic” on page 15-77
- “Inverted Pendulum with Animation” on page 15-78
- “Double Spring Mass System” on page 15-80
- “Tank Fill and Empty with Animation” on page 15-82
- “Simulating Systems with Variable Transport Delay Phenomena” on page 15-85
- “Modeling a Foucault Pendulum” on page 15-90
- “Foucault Pendulum Model with VRML Visualization” on page 15-101
- “Exploring Variable-Step Solvers Using a Stiff Model” on page 15-104
- “Exploring the Solver Jacobian Structure of a Model” on page 15-111
- “Double Bouncing Ball: Use of Adaptive Zero-Crossing Location” on page 15-122
- “Four Hydraulic Cylinder Simulation” on page 15-132
- “Two Cylinder Model with Load Constraints” on page 15-139
- “Modeling a Fault-Tolerant Fuel Control System” on page 15-149
- “Using a Data Dictionary to Manage the Data for a Fuel Control System” on page 15-168
• “Modeling Engine Timing Using Triggered Subsystems” on page 15-172
• “Engine Timing Model with Closed Loop Control” on page 15-184
• “Building a Clutch Lock-Up Model” on page 15-189
• “Modeling Clutch Lock-Up Using If Blocks” on page 15-203
• “Modeling an Anti-Lock Braking System” on page 15-210
• “Automotive Suspension” on page 15-219
• “Modeling an Automatic Transmission Controller” on page 15-226
• “Vehicle Electrical System” on page 15-240
• “Simulating Automatic Climate Control Systems” on page 15-242
• “Vehicle Electrical and Climate Control Systems” on page 15-249
• “Power Window Control Project” on page 15-257
• “Developing the Apollo Lunar Module Digital Autopilot” on page 15-266
• “Designing a High Angle of Attack Pitch Mode Control” on page 15-273
• “Six Degrees of Freedom (6-DoF) Motion Platform” on page 15-291
• “Aircraft Longitudinal Flight Control” on page 15-294
• “Radar Tracking Using MATLAB Function Block” on page 15-297
• “Optical Sensor Image Generation” on page 15-299
• “Air Traffic Control Radar Design” on page 15-311
• “Designing a Guidance System in MATLAB and Simulink” on page 15-318
• “Airframe Trim and Linearize” on page 15-336
• “Anti-Windup Control Using a PID Controller” on page 15-341
• “Bumpless Control Transfer Between Manual and PID Control” on page 15-359
• “Two Degree-of-Freedom PID Control for Setpoint Tracking” on page 15-371
• “Data Typing in Simulink” on page 15-379
• “Data Typing Filter” on page 15-383
• “Simulink Bus Signals” on page 15-385
• “Modeling Arrays of Buses” on page 15-398
• “Matrix Signals” on page 15-400
• “Variable-Size Signal Basic Operations” on page 15-401
• “Variable-Size Signal Length Adaptation” on page 15-403
• “Multimode Variable-Size Signal” on page 15-404
• “Parallel Channel Power Allocation” on page 15-405
• “Merging Signals” on page 15-407
• “Migration to Structure Parameters” on page 15-425
• “Share Data Store Between Instances of a Reusable Algorithm” on page 15-432
• “Attaching Input Data to External Inputs via Custom Input Mappings” on page 15-438
• “Using Mapping Modes with Custom-Mapped External Inputs” on page 15-442
• “Converting Harness-Driven Models to Use Harness-Free External Inputs” on page 15-448
• “Logging States in Structure Format” on page 15-455
• “Saving and Restoring Simulation Operating Point” on page 15-459
• “Logging Intervals” on page 15-464
• “Working with Big Data” on page 15-468
• “Simulink Subsystem Semantics” on page 15-474
• “If-Then-Else Blocks” on page 15-476
• “Triggered Subsystems” on page 15-479
• “Enabled Subsystems” on page 15-482
• “Advanced Enabled Subsystems” on page 15-485
• “Resettable Subsystems” on page 15-489
• “Discrete and Continuous Resettable Subsystems” on page 15-492
• “Variant Subsystems” on page 15-495
• “Block Priority” on page 15-501
• “Monitoring Ink Status on a Shared Printer Using Simulink Functions” on page 15-502
• “Modeling Services and Access from Application Software” on page 15-504
• “Modeling Reusable Components Using Multiply Instanced Simulink Functions” on page 15-506
• “Dynamic Priority Scheduling of Functions” on page 15-509
• “Component-Based Modeling with Model Reference” on page 15-510
• “Viewing Signals in Model Reference Instances” on page 15-515
• “Visualizing Model Reference Architectures” on page 15-525
• “Introduction to Managing Data with Model Reference” on page 15-538
• “Detailed Workflow for Managing Data with Model Reference” on page 15-553
• “Interface Specification Using Bus Objects” on page 15-576
• “Converting Subsystems to Model Reference” on page 15-583
• “Using Data Stores Across Multiple Models” on page 15-587
• “Model Reference Function-Call” on page 15-595
• “Protected Models for Model Reference” on page 15-598
• “Model Reference Variants” on page 15-605
• “Assigning Tasks to Cores for Multicore Programming” on page 15-610
• “Implement an FFT on a Multicore Processor and an FPGA” on page 15-613
• “Multicore Programming of a Field-Oriented Control on Zynq” on page 15-619
• “Multicore Deployment of a Plant Model” on page 15-627
• “Modeling Objects with Identical Dynamics Using For Each Subsystem” on page 15-633
• “Vectorizing a Scalar Algorithm with For Each Subsystem” on page 15-639
• “Tiled Processing of 2D Signals with For Each Subsystem” on page 15-641
• “Neighborhood Processing Using For Each Subsystems” on page 15-643
• “Using a Project with SVN” on page 15-647
• “Using a Project with Git” on page 15-652
• “Using a Project” on page 15-657
• “Perform Impact Analysis with a Project” on page 15-662
• “Referencing Projects from Another Project” on page 15-665
• “Automate Label Management in a Project” on page 15-666
• “Running Custom Tasks with a Project” on page 15-670
• “Upgrade Simulink Models Using a Project” on page 15-674
• “Convert from MDL to SLX in a Project and Preserve Revision History” on page 15-677
• “Export a Subset of a Project Using an Export Profile” on page 15-683
• “Creating Projects Programmatically” on page 15-685
• “Compare and Merge Simulink Models” on page 15-687
• “Compare and Merge Simulink Models Containing Stateflow” on page 15-691
“Resolve Conflicts with Simulink Three-Way Merge” on page 15-695
“Call C Functions Using C Caller Block” on page 15-699
“Bring Custom Image Filter Algorithms as Reusable Blocks in Simulink” on page 15-704
“Custom Code and Hand Coded Blocks using the S-function API” on page 15-707
“Inputs Passed by Value or Address to Legacy Functions” on page 15-708
“Output Passed by Return Argument from Legacy Functions” on page 15-713
“Fixed Point Signals in Legacy Functions” on page 15-716
“Fixed Point Parameters in Legacy Functions” on page 15-719
“Lookup Tables Implemented in Legacy Functions” on page 15-723
“Start and Terminate Actions Within Legacy Functions” on page 15-727
“Using Buses with Legacy Functions Having Structure Arguments” on page 15-732
“Inherited Signal Dimensions for Legacy Function Arguments” on page 15-737
“C++ Object Methods as Legacy Functions” on page 15-741
“Persistent Memory Within Legacy Functions” on page 15-745
“Multi-Dimensional Signals in Legacy Functions” on page 15-749
“Complex Signals in Legacy Function” on page 15-752
“Specified or Inherited Sample Time with Legacy Functions” on page 15-755
“System Identification for an FIR System Using MATLAB System Blocks” on page 15-759
“MATLAB System Block with Variable-Size Input and Output Signals” on page 15-762
“Illustration of Law of Large Numbers” on page 15-764
“Using Buses with MATLAB System Blocks” on page 15-766
“Run Quality Checks on S-Functions” on page 15-768
“Interpolation Algorithm Selection and Trajectory Animation” on page 15-770
“Using the Prelookup and Interpolation Blocks” on page 15-773
“Saving Memory in Prelookup and Interpolation Blocks by Using Smaller Data” on page 15-780
“Model Advisor” on page 15-781
“Simulink Model Discretizer” on page 15-783
“Introduction to Profiling Models” on page 15-784
• “Introduction to Accelerating Models” on page 15-788
• “Determining Why Simulink Accelerator Is Regenerating Code” on page 15-792
• “Parallel Simulations Using Parsim: Test-Case Sweep” on page 15-799
• “Parallel Simulations Using Parsim: Parameter Sweep in Normal Mode” on page 15-804
• “Parallel Simulations Using Parsim: Parameter Sweep in Rapid Accelerator Mode” on page 15-809
• “Rapid Accelerator Simulations Using Parsim” on page 15-814
• “Multiple Simulations Workflow Tips” on page 15-819
• “Streamline Simulink Blockset Authoring Process with Blockset Designer” on page 15-824
• “Importing a Co-Simulation FMU into Simulink” on page 15-826
• “Importing a Model Exchange FMU into Simulink” on page 15-828
• “Using Bus Signals and Structure Parameters in the FMU Import Block” on page 15-830
• “Co-Simulation Signal Compensation” on page 15-833
• “Multithread Co-Simulation” on page 15-836
• “Pulse Width Modulation Using MATLAB System Block” on page 15-838
• “Modeling Cyber-Physical Systems” on page 15-840
• “Schedule an Export-Function Model Using the Schedule Editor” on page 15-847
Simulation of a Bouncing Ball

This example shows how to use two different approaches to modeling a bouncing ball using Simulink®.

Overview

Figure 1: A ball is thrown up with a velocity of 15 m/s from a height of 10 m.

A bouncing ball model is a classic example of a hybrid dynamic system. A hybrid dynamic system is a system that involves both continuous dynamics, as well as, discrete transitions where the system dynamics can change and the state values can jump. The continuous dynamics of a bouncing ball is simply given by:

\[
\frac{dv}{dt} = -g,
\]

\[
\frac{dx}{dt} = v,
\]
where $g$ is the acceleration due to gravity, $x(t)$ is the position of the ball and $v(t)$ is the velocity. Therefore, the system has two continuous states: position $x$ and velocity $v$.

The hybrid system aspect of the model originates from the modeling of a collision of the ball with the ground. If one assumes a partially elastic collision with the ground, then the velocity before the collision, $v^-$, and velocity after the collision, $v^+$, can be related by the coefficient of restitution of the ball, $\kappa$, as follows:

$$v^+ = -\kappa v^-, \quad x = 0$$

The bouncing ball therefore displays a jump in a continuous state (velocity) at the transition condition, $x = 0$.

A bouncing ball is one of the simplest models that shows the **Zeno phenomenon**. Zeno behavior is informally characterized by an infinite number of events occurring in a finite time interval for certain hybrid systems. As the ball loses energy in the bouncing ball model, a large number of collisions with the ground start occurring in successively smaller intervals of time. Hence the model experiences Zeno behavior. Models with **Zeno behavior** are inherently difficult to simulate on a computer, but are encountered in many common and important engineering applications.

**Using Two Integrator Blocks to Model a Bouncing Ball**

Open this model

You can use two Integrator blocks to model a bouncing ball. The Integrator on the left is the velocity integrator modeling the first equation and the Integrator on the right is the position integrator. Navigate to the position integrator block dialog and observe that it has a lower limit of zero. This condition represents the constraint that the ball cannot go below the ground.

The state port of the position integrator and the corresponding comparison result is used to detect when the ball hits the ground and to reset both integrators. The state port of the velocity integrator is used for the calculation of $v^+$. 
Bouncing Ball Model

Two separate integrators are less efficient than a single Second-Order Integrator for simulating a bouncing ball. Click here to see sidemo_bounce for the recommended modeling approach.
To observe the Zeno behavior of the system, navigate to the Solver pane of the Configuration Parameters dialog box. In the 'Zero-crossing options' section, confirm that 'Algorithm' is set to 'Nonadaptive' and that the simulation 'Stop time' is set to 25 seconds. Run the simulation.
Observe that the simulation errors out as the ball hits the ground more and more frequently and loses energy. Consequently, the simulation exceeds the default limit of 1000 for the 'Number of consecutive zero crossings' allowed. Now navigate to the Configuration Parameters dialog box. In the 'Zero-crossing options' section, set the 'Algorithm' to 'Adaptive'. This algorithm introduces a sophisticated treatment of such chattering behavior. Therefore, you can now simulate the system beyond 20 seconds. Note, however, the chatter of the states between 21 seconds and 25 seconds and warning from Simulink about the strong chattering in the model around 20 seconds.

**Using a Second-Order Integrator Block to Model a Bouncing Ball**

Open this model

You can use a single Second-Order Integrator block to model this system. The second equation \( \frac{dx}{dt} = v \) is internal to the Second-Order Integrator block in this case. Navigate to the Second-Order Integrator block dialog and notice that, as earlier, \( x \) has a lower limit of zero. Navigate to the Attributes tab on the block dialog and note that the option 'Reinitialize dx/dt when x reaches saturation' is checked. This parameter allows us to reinitialize \( \frac{dx}{dt} \) (\( v \) in the bouncing ball model) to a new value at the instant \( x \) reaches its saturation limit. For the bouncing ball model, this option therefore implies that when the ball hits the ground, its velocity can be set to a different value, i.e., to the velocity after the impact. Notice the loop for calculating the velocity after a collision with the ground.

To capture the velocity \( v^- \) of the ball just before the collision, the \( \frac{dx}{dt} \) output port of the Second-Order Integrator block and a Memory block are used. \( v^- \) is then used to calculate the rebound velocity \( v^+ \).
Bouncing Ball Model

Gravitational acceleration

Initial Velocity

Second-Order Integrator

Coefficient of Restitution

Copyright 2004-2013 The MathWorks, Inc.
Navigate to the Solver pane of the Configuration Parameters dialog box. Confirm that 'Algorithm' is set to 'Nonadaptive' in the 'Zero-crossing options' section and the simulation 'Stop Time' is set to 25 seconds. Simulate the model. Note that the simulation encountered no problems. You were able to simulate the model without experiencing excessive chatter after $t = 20$ seconds and without setting the 'Algorithm' to 'Adaptive'.
Second-Order Integrator Model Is the Preferable Approach to Modeling Bouncing Ball

One can analytically calculate the exact time $t^*$ when the ball settles down to the ground with zero velocity by summing the time required for each bounce. This time is the sum of an infinite geometric series given by:

$$ t^* = \frac{1}{g} \left( v_0 + v_1 \left( \frac{1 + \kappa}{1 - \kappa} \right) \right), \quad v_1 = \sqrt{\frac{v_0^2}{g} + 2g x_0}. $$

Here $x_0$ and $v_0$ are initial conditions for position and velocity respectively. The velocity and the position of the ball must be identically zero for $t > t^*$. In the figure below, results from both simulations are plotted near $t^*$. The vertical red line in the plot is $t^*$ for the given model parameters. For $t < t^*$ and far away from $t^*$, both models produce accurate and identical results. Hence, only a magenta line from the second model is visible in the plot. However, the simulation results from the first model are inexact after $t^*$; it continues to display excessive chattering behavior for $t > t^*$. In contrast, the second model using the Second-Order Integrator block settles to exactly zero for $t > t^*$. 
Figure 2: Comparison of simulation results from the two approaches.

Figure 2 conclusively shows that the second model has superior numerical characteristics as compared to the first model. The reason for the higher accuracy associated with the Second-Order Integrator model is as follows. The second differential equation \( \frac{dx}{dt} = v \) is internal to the Second-Order Integrator block. Therefore, the block algorithms can leverage this known relationship between the two states and deploy heuristics to clamp down the undesirable chattering behavior for certain conditions. These heuristics become active when the two states are no longer mutually consistent with each other due to integration errors and chattering behavior. You can thus use physical knowledge of the
system to alleviate the problem of simulation getting stuck in a Zeno state for certain classes of Zeno models.
Single Hydraulic Cylinder Simulation

This example shows how to use Simulink® to model a hydraulic cylinder. You can apply these concepts to applications where you need to model hydraulic behavior. See two related examples that use the same basic components: four cylinder model and two cylinder model with load constraints.

- **Note:** This is a basic hydraulics example. You can more easily build hydraulic and automotive models using Simscape™ Driveline™ and Simscape Fluids™.

- **Simscape Fluids** provides component libraries for modeling and simulating fluid systems. It includes models of pumps, valves, actuators, pipelines, and heat exchangers. You can use these components to develop fluid power systems such as front-loader, power steering, and landing gear actuation systems. Engine cooling and fuel supply systems can also be developed with Simscape Fluids. You can integrate mechanical, electrical, thermal, and other systems using components available within the Simscape product family.

- **Simscape Driveline** provides component libraries for modeling and simulating one-dimensional mechanical systems. It includes models of rotational and translational components, such as worm gears, planetary gears, lead screws, and clutches. You can use these components to model the transmission of mechanical power in helicopter drivetrains, industrial machinery, vehicle powertrains, and other applications. Automotive components, such as engines, tires, transmissions, and torque converters, are also included.

**Analysis and Physics of the Model**

Figure 1 shows a schematic diagram of the basic model. The model directs the pump flow, \( Q \), to supply pressure, \( p_1 \), from which laminar flow, \( q_{1\text{ex}} \), leaks to exhaust. The control valve for the piston/cylinder assembly is modeled as turbulent flow through a variable-area orifice. Its flow, \( q_{12} \), leads to intermediate pressure, \( p_2 \), which undergoes a subsequent pressure drop in the line connecting it to the actuator cylinder. The cylinder pressure, \( p_3 \), moves the piston against a spring load, resulting in position \( x \).
Figure 1: Schematic diagram of the basic hydraulic system

At the pump output, the flow is split between leakage and flow to the control valve. We model the leakage, $q_{1ex}$, as laminar flow (see Equation Block 1).

**Equation Block 1**

\[
Q = q_{12} + q_{1ex}
\]

\[
q_{1ex} = C_2 \cdot p_1
\]

\[
p_1 = \frac{(Q - q_{12})}{C_2}
\]

$Q =$ pump flow

$q_{12} =$ control valve flow

$q_{1ex} =$ leakage

$C_2 =$ flow coefficient

$p_1 =$ pump pressure

We modeled turbulent flow through the control valve with the orifice equation. The sign and absolute value functions accommodate flow in either direction (see Equation Block 2).
Equation Block 2

\[ q_{12} = C_d \cdot A \cdot \text{sgn}(p_1 - p_2) \cdot \sqrt{\frac{2}{\rho} |p_1 - p_2|} \]

- \( C_d \) = orifice discharge coefficient
- \( A \) = orifice area
- \( p_2 \) = pressure downstream of control valve
- \( \rho \) = fluid density

The fluid within the cylinder pressurizes due to this flow, \( q_{12} = q_{23} \), minus the compliance of the piston motion. We also modeled fluid compressibility in this case (see Equation Block 3).

Equation Block 3

\[ \frac{dp_3}{dt} = \frac{\beta}{V_3} \left( q_{12} - A_c \frac{dx}{dt} \right) \]

- \( V_3 = V_{30} + A_c \cdot x \)
- \( p_3 \) = piston pressure
- \( \beta \) = fluid bulk modulus
- \( V_3 \) = fluid volume at \( p_3 \)
- \( V_{30} \) = fluid volume in the piston for \( x = 0 \)
- \( A_c \) = cylinder cross-sectional area

We neglected the piston and spring masses because of the large hydraulic forces. We completed the system of equations by differentiating this relationship and incorporating the pressure drop between \( p_2 \) and \( p_3 \). Equation Block 3 models laminar flow in the line from the valve to the actuator. Equation block 4 gives the force balance at the piston.

Equation Block 4
Figure 2 shows the top level diagram of the model. The pump flow and the control valve orifice area are simulation inputs. The model is organized as two subsystems: the 'Pump' and the 'Valve/Cylinder/Piston/Spring Assembly'.

**Modeling**

To open this model, type `sldemo_hydcyl` at MATLAB® terminal (click on the hyperlink if you are using MATLAB Help). Press the "Play" button on the model toolbar to run the simulation.

- Note: The model logs relevant data to MATLAB workspace, into the Simulink.SimulationOutput object `out`. The signal logging data is stored within `out`, in a structure called `sldemo_hydcyl_output`. Logged signals have a blue indicator (see the model). Read more about Signal Logging in Simulink Help.

\[ x = p_3 \frac{A_c}{K} \]

\[ \frac{dx}{dt} = \frac{dp_3}{dt} \frac{A_c}{K} \]

\[ q_{23} = q_{12} = C_1 (p_2 - p_3) \]

\[ p_2 = p_3 + \frac{q_{12}}{C_1} \]

\[ K = \text{spring constant} \]

\[ C_1 = \text{laminar flow coefficient} \]
Single Hydraulic Cylinder Simulation

Copyright 1990-2012 The MathWorks, Inc.
Figure 2: Single cylinder model and simulation results

'Pump' Subsystem

Right click on the 'Pump' masked subsystem and select "Look Under Mask" to see its components. The pump model computes the supply pressure as a function of the pump flow and the load (output) flow (Figure 3). \( Q_{\text{pump}} \) is the pump flow data (saved in the
A matrix with column vectors of time points and the corresponding flow rates \([T, Q]\) specifies the flow data. The model calculates pressure \(p_1\) as indicated in Equation Block 1. Because \(Q_{out} = q_{12}\) is a direct function of \(p_1\) (via the control valve), an algebraic loop is formed. An estimate of the initial value, \(p_{10}\), enables a more efficient solution.

**Figure 3:** The pump subsystem

We masked the 'Pump' subsystem in Simulink to allow the user to easily access the parameters (see Figure 4). The parameters to be specified are \(T, Q, p_{10},\) and \(C_2\). We then assigned the masked block the icon shown in Figure 2, and saved it in a Simulink library.
Figure 4: Entering pump parameters

'Valve/Cylinder/Piston/Spring Assembly' Subsystem

Right click on the 'Valve/Cylinder/Piston/Spring Assembly' in the model and select "Look Under Mask" to see the Actuator subsystem (see Figure 5). A system of differential-algebraic equations models the cylinder pressurization with the pressure $p_3$, which appears as a derivative in Equation Block 3 and is used as the state (integrator). If we
neglect piston mass, the spring force and piston position are direct multiples of $p_3$ and the velocity is a direct multiple of $p_3$'s time derivative. This latter relationship forms an algebraic loop around the 'Beta' Gain block. The intermediate pressure $p_2$ is the sum of $p_3$ and the pressure drop due to the flow from the valve to the cylinder (Equation Block 4). This relationship also imposes an algebraic constraint through the control valve and the $1/C_1$ gain.

The control valve subsystem computes the orifice (Equation Block 2). It uses as inputs the upstream and downstream pressures and the variable orifice area. The 'Control Valve Flow' Subsystem computes the signed square root:

$$y = \text{sgn}(u) \sqrt{|u|}$$

Three nonlinear functions are used, two of which are discontinuous. In combination, however, $y$ is a continuous function of $u$.

---

**Figure 5:** The valve/cylinder/piston/spring subsystem
Results

Simulation Parameters

We simulated the model using the following data. The information is loaded from a MAT-file - sldemo_hydcyl_data.mat, which is also used for the other two hydraulic cylinder models. The users can enter data via the Pump and Cylinder Masks shown in Figures 4 and 6.

\[ T = [0 \ 0.04 \ 0.04 \ 0.05 \ 0.05 \ 0.1] \text{ sec} \]
\[ Q = [0.005 \ 0.005 \ 0 \ 0 \ 0.005 \ 0.005] \text{ m}^3/\text{sec} \]
A hydraulic cylinder is fed through a variable-area control valve, producing an axial piston force which compresses a linear spring.

Inputs:
- \( A = \text{valve flow area} \)
- \( \text{vector} \)
- \( p_1 = \text{supply pressure} \)

Outputs:
- \( p = \text{valve and piston pressure} \)
- \( x = \text{piston displacement} \)
- \( q_{in} = \text{input flow rate} \)

See "Using Simulink and Stateflow in Automotive Applications" for a mathematical derivation of the subsystem operation.

Parameters:
- \( \text{Spring rate:} \)
  - \( K \)

- \( \text{Flow gain from valve to actuator:} \)
  - \( C_1 \)

- \( \text{Actuator cross-sectional area:} \)
  - \( A_c \)

- \( \text{Discharge coefficient:} \)
  - \( C_d \)

- \( \text{Fluid density:} \)
  - \( \rho \)

- \( \text{Cylinder volume at } x = 0: \)
  - \( V_{30} \)

- \( \text{Bulk modulus:} \)
  - \( \beta \)

Assembly name:
- \( \text{Spring} \)
Figure 6: Entering valve/cylinder/piston/spring assembly parameters

Plotting Simulation Results

The system initially steps to a pump flow of $0.005 \text{ m}^3/\text{sec} = 300 \text{ l/min}$, abruptly steps to zero at $t=0.04 \text{ sec}$, then resumes its initial flow rate at $t=0.05 \text{ sec}$.

The control valve starts with zero orifice area and ramps to $1e-4 \text{ sq.m.}$ during the $0.1 \text{ sec}$ simulation time. With the valve closed, all of the pump flow goes to leakage so the initial pump pressure increases to $p_{10} = \frac{Q}{C_2} = 1667 \text{ kPa}$.

As the valve opens, pressures $p_2$ and $p_3$ build up while $p_1$ decreases in response to the load increase as shown in Figure 7. When the pump flow cuts off, the spring and piston act like an accumulator and $p_3$ decreases continuously. Then the flow reverses direction, so $p_2$, though relatively close to $p_3$, falls abruptly. At the pump itself, all of the back-flow leaks and $p_1$ drops radically. The behavior reverses as the flow is restored.

The piston position is directly proportional to $p_3$, where the hydraulic and spring forces balance. Discontinuities in the velocity at $0.04 \text{ sec}$ and $0.05 \text{ sec}$ indicate negligible mass. The model reaches a steady state when all of the pump flow again goes to leakage, now due to zero pressure drop across the control valve (which means $p_3 = p_2 = p_1 = p_{10}$).
**Figure 7:** Simulation Results: System Pressures
Figure 8: Simulation Results: Hydraulic Cylinder Piston Position

Closing the Model

Close the model and clear generated data.
Thermal Model of a House

This example shows how to use Simulink® to create the thermal model of a house. This system models the outdoor environment, the thermal characteristics of the house, and the house heating system.

The sldemo_househeat_data.m file initializes data in the model workspace. To make changes, you can edit the model workspace directly or edit the file and re-load the model workspace. To view the model workspace, select View > Model Explorer from the Simulink editor.

Opening the Model

Open the sldemo_househeat model

Figure 1: The House Heating Model
Model Initialization

This model calculates heating costs for a generic house. When the model is opened, it loads the information about the house from the sldemo_househeat_data.m file. The file does the following:

- Defines the house geometry (size, number of windows)
- Specifies the thermal properties of house materials
- Calculates the thermal resistance of the house
- Provides the heater characteristics (temperature of the hot air, flow-rate)
- Defines the cost of electricity (0.09$/kWhr)
- Specifies the initial room temperature (20 deg. Celsius = 68 deg. Fahrenheit)
- **Note:** Time is given in units of hours. Certain quantities, like air flow-rate, are expressed per hour (not per second).

Model Components

Set Point

"Set Point" is a constant block. It specifies the temperature that must be maintained indoors. It is 70 degrees Fahrenheit by default. Temperatures are given in Fahrenheit, but then are converted to Celsius to perform the calculations.

Thermostat

"Thermostat" is a subsystem that contains a Relay block. The thermostat allows fluctuations of 5 degrees Fahrenheit above or below the desired room temperature. If air temperature drops below 65 degrees Fahrenheit, the thermostat turns on the heater. See the thermostat subsystem below.

Open the Thermostat subsystem
Figure 2: The "Thermostat" Subsystem

Heater

"Heater" is a subsystem that has a constant air flow rate, "Mdot", which is specified in the sldemo_househeat_data.m file. The thermostat signal turns the heater on or off. When the heater is on, it blows hot air at temperature $T_{heater}$ (50 degrees Celsius = 122 degrees Fahrenheit by default) at a constant flow rate of Mdot (1kg/sec = 3600kg/hr by default). The heat flow into the room is expressed by the Equation 1.

**Equation 1**

\[
\frac{dQ}{dt} = (T_{heater} - T_{room}) \cdot Mdot \cdot c
\]

- $\frac{dQ}{dt}$ = heat flow from the heater into the room
- $c$ = heat capacity of air at constant pressure
- $Mdot$ = air mass flow rate through heater (kg/hr)
- $T_{heater}$ = temperature of hot air from heater
- $T_{room}$ = current room air temperature

Open the Heater subsystem
Figure 3: The Heater Subsystem

Cost Calculator

"Cost Calculator" is a Gain block. "Cost Calculator" integrates the heat flow over time and multiplies it by the energy cost. The cost of heating is plotted in the "PlotResults" scope.

House

"House" is a subsystem that calculates room temperature variations. It takes into consideration the heat flow from the heater and heat losses to the environment. Heat losses and the temperature time derivative are expressed by Equation 2.

Equation 2

\[
\left(\frac{dQ}{dt}\right)_{\text{losses}} = \frac{T_{\text{room}} - T_{\text{out}}}{R_{\text{eq}}}
\]

\[
\frac{dT_{\text{room}}}{dt} = \frac{1}{M_{\text{air}} \cdot c} \cdot \left( \frac{dQ_{\text{heater}}}{dt} - \frac{dQ_{\text{losses}}}{dt} \right)
\]

\[M_{\text{air}} = \text{mass of air inside the house}\]

\[R_{\text{eq}} = \text{equivalent thermal resistance of the house}\]

Open the House subsystem
Figure 4: The House Subsystem

Modeling the Environment

We model the environment as a heat sink with infinite heat capacity and time varying temperature Tout. The constant block "Avg Outdoor Temp" specifies the average air temperature outdoors. The "Daily Temp Variation" Sine Wave block generates daily temperature fluctuations of outdoor temperature. Vary these parameters and see how they affect the heating costs.

Running the Simulation and Visualizing the Results

Run the simulation and visualize the results. Open the "PlotResults" scope to visualize the results. The heat cost and indoor versus outdoor temperatures are plotted on the scope. The temperature outdoor varies sinusoidally, whereas the indoors temperature is maintained within 5 degrees Fahrenheit of "Set Point". Time axis is labeled in hours.
According to this model, it would cost around $30 to heat the house for two days. Try varying the parameters and observe the system response.

**Remarks**

This particular model is designed to calculate the heating costs only. If the temperature of the outside air is higher than the room temperature, the room temperature will exceed the desired "Set Point".

You can modify this model to include an air conditioner. You can implement the air conditioner as a modified heater. To do this, add parameters like the following to sldemo_househeat_data.m.
• Cold air output
• Temperature of the stream from the air conditioner
• Air conditioner efficiency

You would also need to modify the thermostat to control both the air conditioner and the heater.
Approximating Nonlinear Relationships: Type S Thermocouple

This example shows how to approximate nonlinear relationships of a type S thermocouple.

Thermocouple Modeling and Signal Conversion

The thermocouple is one of the popular analog transducers today, along with other devices such as position sensors, strain gages, pressure transducers, and resistance temperature devices (RTDs). Operating under the principle of the Seebeck effect (a.k.a. thermoelectric effect), thermocouples have an empirically determined nonlinear behavior that is well known over each junction type's useful operating range. If you run the model, you will be able to see the cumulative effect of each component on dynamic measurement accuracy. This example will focus on models for each of these components in a dynamic temperature measurement system for a Type S (Platinum-10% Rhodium(+) versus Platinum(-)) : a thermocouple device and probe assembly, a signal conditioning method, an analog to digital converter (ADC), and a software specification for converting the ADC output into a temperature value. An additional section shows how to obtain and use standard NIST ITS-90 thermocouple data with Simulink® models. Look-up tables and a polynomial block are used in this design to capture the nonlinear behavior of the thermocouple. Note that the polynomial block is a viable alternative to look-up tables for some applications, minimizing ROM usage at the cost of some additional computation for the full polynomial representation.

Open the model

open_system('sldemo_tc')
Approximating Nonlinear Relationships: Type S Thermocouple

**Temperature measurement system**

A chain of components from physical phenomenon to software values

1) The thermocouple model has a polynomial for a Type S thermocouple using data from NIST Standard Database 60 as well as a simple model of the probe and bead time constant.

2) The analog to digital converter (ADC) subsystem includes analog scaling and filtering, plus a sampled data quantizer. Conversion time is not modeled as it is typically ~2 orders of magnitude smaller in time scale compared to the rest of the simulation.

3) Two conversion techniques are used in the software specification to show some of the options available for modeling nonlinear behavior.

**Figure 1:** Temperature measurement system: a chain of components from physical phenomenon to software values

**Simulating the Thermocouple Signal**

The two main features of the thermocouple model are the probe and bead dynamics and the thermocouple's conversion of temperature into a millivolt signal. The probe+bead dynamics are modeled as a 30 msec first order system, and the nonlinear thermocouple behavior is modeled using the segment 1 polynomial data from NIST Standard Database 60 for a Type S thermocouple from -50 to 1063 degC. For numerical stability, the coefficients were scaled to return microvolts from the polynomial block. The output of the 1 Type S Thermocouple model subsystem is then converted to volts with a Unit
Conversion block. Note that units are specified on the subsystem input and output ports and displayed on the subsystem icons. To learn more about units in Simulink see Simulink Units.

An alternative implementation to using the polynomial is an interpolated look-up table. Table data could be used in a look-up table block in place of the polynomial block. Sample data was constructed from NIST Standard Database 60 for a Type S thermocouple in file sldemo_create_tc_tabledata.m. Access to this database is described below in the section titled "Thermocouple Reference Data Download and Import Procedure".

**Anti-Aliasing Filter and Analog to Digital Converter (ADC) Models**

The ADC in this model expects a 0 to 5 volt signal, so the raw thermocouple sense voltage is biased and amplified corresponding to a range of -0.235 mV to 18.661 mV (-50 degC to 1765 degC). A third order Butterworth anti-aliasing filter was designed for \( W_n = 15 \) Hz using the Signal Processing Toolbox™:

\[
[num, den] = butter(3, 15*2*pi, 'low', 's')
\]

The output of the anti-aliasing filter feeds a sample-and-hold device that drives the quantization model. Since the sample period is 20 msec in this example, the conversion time is ignored as it is typically 2 orders of magnitude smaller for devices currently available. (Note: if the conversion time were an appreciable fraction of the sample period, it could not be ignored as it would significantly affect the system's dynamics.)

The quantization algorithm in the model takes in a 0 to 5 volt signal and outputs a 12-bit digital word in a 16-bit signed integer. A value of 0 corresponds to 0 Volts and a value of 4096 would correspond to 5 Volts. A change in the least significant bit (LSB) is about 1.2 mV. As 12 bits can only reach the value of 4095, the highest voltage that can be read by this device is approximately 4.9988 Volts. In order to have no more than 1/2 LSB error within the operating range, the quantizer changes values midway between each voltage point, resulting in a 1/2-width interval at 0 Volts and a 3/2-width interval just below 5 Volts. The last interval has 1 full LSB due to its 3/2-width size.

**Understanding Data Converters**

The sldemo_adc_quantize model allows you to explore the A/D converter component in more detail:

open_system('sldemo_adc_quantize')
Figure 2: Details of ADC quantization modeling (zero conversion time)

sim('sldemo_adc_quantize')
set(gcf,'Color',[1,1,1]);
Figure 3: Quantization characteristic of ADC.

```matlab
ax = get(gcf,'Children');
set(ax(1), 'xlim', [4085, 4100]);
```
Software Specification for Converting ADC Output to Temperature Values

The input conversion subsystem requires a 16 bit unsigned integer input from the ADC whose full scale range is 0 to 4095 counts, corresponding to -0.235 mV and 18.6564 mV thermocouple loop voltage. The best accuracy and fastest algorithm for input conversion is a direct look-up table. Since the input is an integer from 0 to 4095, a table can be constructed that gives the thermocouple temperature corresponding to each possible input value, so the conversion process can be reduced to indexing into a table. This however, requires one number per ADC output value and for a 12-bit ADC, this can be a burden in memory constrained environments. For double precision data, this is a 16 kB
ROM requirement. See file `sldemo_create_tc_tabledata.m` for the method used to construct the direct look-up table from the Type S thermocouple reference data. The error associated with this approach is entirely isolated to the table construction process as there is an output value associated with every possible input value - the run-time look-up process introduces no additional error.

An interpolated table was also put into the model, using only 664 bytes. This is a big reduction in ROM required compared to the direct table look-up, but it takes a bit longer to compute than an indirect memory access and introduces error into the measurement, which goes down as the number of points in the table increases.

```
open_system(sprintf('sldemo_tc/3 Software specification\nfor converting\nADC values to
1-D T[k] (°C)
1-D T(u) (°C)
 Tipo S direct conversion
(Types_tdegc_vs_bits)

Input: 0 to 4095 integer "counts", where
0 counts corresponds to 0 input volts and -0.236 thermocouple millivolts
4096 counts corresponds to 5 input volts and 18.661 thermocouple millivolts
```

---

**Tables derived from ITS-90 thermocouple reference data from NIST Standard Database 60 related to NIST Monograph 175 (ITS-90).**

---

**Thermocouple Reference Data Download and Import Procedure**

Using the NIST ITS-90 Thermocouple Database (Standard Reference Database 60 from NIST Monograph 175), you can access the standard reference data describing the
behavior for the eight standard thermocouple types. This database relates thermocouple output to the International Temperature Scale of 1990. Follow these steps to acquire and read in the data needed to fully exercise the support files included with this example:

1. Visit the NIST Standard database 60 site on the Internet and download the file all.tab to a local directory. This file is the one under the All Thermocouple Types hyperlink. After the download is complete, return to this page.

2. cd to the directory where you downloaded the all.tab thermocouple database

3. Parse the database and convert it to a MATLAB structure array using the conversion tool readstdtcdata.m:

   tcdata = readstdtcdata('all.tab');
save thermocouple_its90.mat tcdata;

(tip: highlight the above MATLAB code and use right mouse menuitem "Evaluate Selection" to execute it)

You now have a complete set of temperature (T, degC) vs. voltage (E, mV) data, approximating polynomial coefficients, and inverse polynomial coefficients for the standard thermocouple types B, E, J, K, N, R, S, and T in the tcdata variable of file thermocouple_its90.mat. The MATLAB script in sldemo_create_tc_tabledata.m uses this data to prepare lookup table block parameters used in the example model.

**Conditioning Reference Data for Use in a Look-Up Table**

If you review the thermocouple data tables in the tcdata structure or in all.tab, you will probably notice a few things:

- Repeated 0 degC temperature points in the data
- Repeated voltage points in the data due to the 3-digit output format
- Non-monotonic behavior with two temperatures having the same output, e.g., for the Type B thermocouple, T(E) won't work but E(T) does work

The readstdtcdata() routine will remove the repeated 0 degC temperature points from the data, but the repeated voltages due to the data format and non-monotonic behavior of some of the curve ends must be dealt with on an individual basis. A reference model named sldemo_tc_blocks was constructed using sldemo_tc_blocks_data.mat. It contains look-up tables with data populated from the interpolating polynomials for the eight standard thermocouples in all.tab:
open_system('sldemo_tc_blocks')

**Figure 5:** Full-Range 2 degrees Celsius Interpolated Thermocouple Tables created from ITS-90 Interpolating Polynomials

**References**

1. NIST ITS-90 Thermocouple Database URL: https://srdata.nist.gov/its90/main


3. The International Temperature Scale of 1990 (ITS-90), Consultative Committee for Thermometry (CCT) of the International Committee for Weights and Measures (CIPM)

NOTE: for determining empirical relationships of complex systems such as engines and for fitting models to measured data, MathWorks® offers the Model Based Calibration Toolbox which employs the Design of Experiments methodology for optimized table database creation, plus value extraction and automated table filling components.
This example shows some of the main steps needed to design and evaluate a sine wave data table for use in digital waveform synthesis applications in embedded systems and arbitrary waveform generation instruments.

Real-Time direct digital synthesis of analog waveforms using embedded processors and digital signal processors (DSPs) connected to digital-to-analog converters (DACs) is becoming pervasive even in the smallest systems. Developing waveforms for use in embedded systems or laboratory instruments can be streamlined using the tight integration of MATLAB® and Simulink®. You can develop and analyze the waveform generation algorithm and its associated data at your desktop before implementing it with Simulink® Coder™ on target hardware.

When feasible, the most accurate way to digitally synthesize a sine wave is to compute the full precision sin() function directly for each time step, folding omega*t into the interval 0 to 2*pi. In real-time systems, the computational burden is typically too large to permit this approach. One popular way around this obstacle is to use a table of values to approximate the behavior of the sin() function, either from 0 to 2*pi, or even half wave or quarter wave data to leverage symmetry.

Tradeoffs to consider include algorithm efficiency, data ROM size required, and accuracy/spectral purity of the implementation. Similar analysis is needed when performing your own waveform designs. The table data and look-up algorithm alone do not determine performance in the field. Additional considerations such as the accuracy and stability of the real-time clock, and digital to analog converter are also needed in order to assess overall performance. The Signal Processing Toolbox™ and the DSP System Toolbox™ complement the capabilities of MATLAB and Simulink for work in this area.

Another popular way to approximate the behavior of sine wave is to use the CORDIC approximation method. CORDIC is an acronym for COordinate Rotation DIgital Computer. The Givens rotation-based CORDIC algorithm is among one of the most hardware-efficient algorithms because it requires only shift-add iterative operations. If you have Fixed-Point Designer license then you can see fi_sin_cos_demo for a tutorial style example on CORDIC approximation.

The distortion analysis in this example is based on principles presented in "Digital Sine-Wave Synthesis Using the DSP56001/DSP56002", by Andreas Chrysafis, Motorola® Inc. 1988
Create a Table in Double Precision Floating Point

The following commands make a 256 point sine wave and measure its total harmonic distortion when sampled first on the points and then by jumping with a delta of 2.5 points per step using linear interpolation. Similar computations are done by replacing the sine values with CORDIC sine approximation. For frequency-based applications, spectral purity can be more important than absolute error in the table.

The file ssinthd.m is the core function in this example. It is used for calculating total harmonic distortion (THD) for digital sine wave generation with or without interpolation. This THD algorithm proceeds over an integral number of waves to achieve accurate results. The number of wave cycles used is A. Since the step size 'delta' is A/B and traversing A waves will hit all points in the table at least one time, which is needed to accurately find the average THD across a full cycle.

The relationship used to calculate THD is:

\[
\text{THD} = \frac{\text{ET} - \text{EF}}{\text{ET}}
\]

where ET = total energy, and EF = fundamental energy

The energy difference between ET and EF is spurious energy.

\[
\begin{align*}
N &= 256; \\
\text{angle} &= 2\pi \times (0:(N-1))/N; \\
\text{s} &= \sin(\text{angle})'; \\
\text{thd} \_\text{ref} \_1 &= \text{ssinthd}(\text{s}, 1, N, 1, 'direct') \\
\text{thd} \_\text{ref} \_2p5 &= \text{ssinthd}(\text{s}, 5/2, 2*N, 5, 'linear') \\
\text{cs} &= \text{cordicsin}(\text{angle}, 50)'; \\
\text{thd} \_\text{ref} \_1\text{c} &= \text{ssinthd}(\text{cs}, 1, N, 1, 'direct') \\
\text{thd} \_\text{ref} \_2p5\text{c} &= \text{ssinthd}(\text{cs}, 5/2, 2*N, 5, 'linear') \\
\end{align*}
\]

\[
\begin{align*}
\text{thd} \_\text{ref} \_1 &= 4.8049e-32 \\
\text{thd} \_\text{ref} \_2p5 &= 1.4176e-09
\end{align*}
\]
thd_ref_1c =
    1.2577e-30

thd_ref_2p5c =
    1.4176e-09

Put the Sine Wave Approximations in a Model

You can put the sine wave designed above into a Simulink model and see how it works as a direct lookup, with linear interpolation, and with CORDIC approximation. This model compares the output of the floating point tables to the sin() function. As expected from the THD calculations, the linear interpolation has a lower error than the direct table lookup in comparison to the sin() function. The CORDIC approximation shows a lower error margin when compared to the linear interpolation method. This margin depends on the number of iterations when computing the CORDIC sin approximation. You can typically achieve greater accuracy by increasing the number of iterations (corresponding to a longer computation time). The CORDIC approximation eliminates the need for explicit multipliers. It is used when multipliers are less efficient or non-existent in hardware.

Open the sldemo_tonegen model

open_system('sldemo_tonegen');
set_param('sldemo_tonegen', 'StopFcn','');
out = sim('sldemo_tonegen');
currentFig = figure('Color',[1,1,1]);
subplot(3,1,1), plot(out.tonegenOut.time, out.tonegenOut.signals(1).values); grid
    title({'Difference between direct look-up', 'and reference signal'});
subplot(3,1,2), plot(out.tonegenOut.time, out.tonegenOut.signals(2).values); grid
    title({'Difference between interpolated look-up', 'and reference signal'});
subplot(3,1,3), plot(out.tonegenOut.time, out.tonegenOut.signals(3).values); grid
    title({'Difference between CORDIC sine', 'and reference signal'});
Comparison in approximate sine wave accuracy between direct look-up, linear interpolation and CORDIC approximation vs. reference signal

Copyright 1990-2013 The MathWorks, Inc.
Taking a Closer Look at Waveform Accuracy

Zooming in on the signals between 4.8 and 5.2 seconds of simulation time (for example), you can see a different characteristic due to the different algorithms used:

```matlab
ax = get(currentFig,'Children');
set(ax(3),'xlim',[4.8, 5.2])
set(ax(2),'xlim',[4.8, 5.2])
set(ax(1),'xlim',[4.8, 5.2])
```
The Same Table, Implemented in Fixed Point

Now convert the floating point table into a 24 bit fractional number using 'nearest' rounding. The new table is tested for total harmonic distortion in direct lookup mode at 1, 2, and 3 points per step, then with fixed point linear interpolation.

```matlab
bits = 24;
is   = num2fixpt( s, sfrac(bits), [], 'Nearest', 'on');
thd_direct1 = ssinthd(is, 1, N, 1, 'direct')
thd_direct2 = ssinthd(is, 2, N, 2, 'direct')
thd_direct3 = ssinthd(is, 3, N, 3, 'direct')

thd_linterp_2p5 = ssinthd(is, 5/2, 2*N, 5, 'fixptlinear')
```
thd_direct1 =  
2.6423e-15

thd_direct2 =  
2.8660e-15

thd_direct3 =  
2.6423e-15

thd_linterp_2p5 =  
1.4175e-09

**Compare Results for Different Tables and Methods**

Choosing a table step rate of 8.25 points per step (33/4), jump through the double precision and fixed point tables in both direct and linear modes and compare distortion results:

```
thd_double_direct = ssinthd( s, 33/4, 4*N, 33, 'direct')
```

```
= ssinthd(is, 33/4, 4*N, 33, 'direct')
```

```
thd_double_linear = ssinthd( s, 33/4, 4*N, 33, 'linear')
```

```
= ssinthd(is, 33/4, 4*N, 33, 'fixptlinear')
```

thd_double_direct =  
4.7061e-05

thd_sfrac24_direct =  
4.7061e-05

thd_double_linear =  

Using Preconfigured Sine Wave Blocks

Simulink also includes a Sine Wave source block with continuous and discrete modes, plus fixed point Sin and Cosine function blocks that implement the function approximation with a linearly interpolated lookup table that exploits the quarter wave symmetry of sine and cosine. Example sldemo_tonegen_fixpt uses a sampled sine wave source as the reference signal and compares it with a lookup table with or without interpolation, and with CORDIC sine approximation in fixed point data types.

Open the sldemo_tonegen_fixpt model

```matlab
open_system('sldemo_tonegen_fixpt');
set_param('sldemo_tonegen_fixpt', 'StopFcn','');
out = sim('sldemo_tonegen_fixpt');
figure('Color',[1,1,1]);
subplot(3,1,1), plot(out.tonegenOut.time, out.tonegenOut.signals(1).values); grid
title({'Difference between direct look-up', 'and reference signal'});
subplot(3,1,2), plot(out.tonegenOut.time, out.tonegenOut.signals(2).values); grid
title({'Difference between interpolated look-up', 'and reference signal'});
subplot(3,1,3), plot(out.tonegenOut.time, out.tonegenOut.signals(3).values); grid
title({'Difference between CORDIC sine', 'and reference signal'});
```
Additional blocks to calculate sine and approximate sine using various algorithms.

Copyright 1990-2013 The MathWorks, Inc.
Use of Sine Function with a Clock Input

The model also compares the sine wave source reference with the sin() function whose input angle in radians is time based (computed using a clock). This section tests the assumption that the clock input would return repeatable results from the sin() function for period 2*pi. The following plot shows that the sin() function accumulates error when its input is time based. This also shows that a sampled sine wave source is more accurate to use as a waveform generator.

```matlab
subplot(1,1,1), plot(out.tonegenOut.time, out.tonegenOut.signals(4).values); grid
title({['Difference between time based sin()', 'and reference signal']});
```
Survey of Behavior for Direct Lookup and Linear Interpolation

The file sldemo_sweepetable_thd.m performs a full frequency sweep of the fixed point tables and will let us more thoroughly understand the behavior of this design. Total harmonic distortion of the 24-bit fractional fixed point table is measured at each step size, moving through it D points at a time, where D is a number from 1 to N/2, incrementing by 0.25 points. N is 256 points in this example; the 1, 2, 2.5, and 3 cases were done above. Frequency is discrete and therefore a function of the sample rate.

Notice the modes of the distortion behavior in the plot. They match with common sense: when retrieving from the table precisely at a point, the error is smallest; linear interpolation has a smaller error than direct lookup in between points. What is not
intuitive is that the error is relatively constant for each of the modes up to the Nyquist frequency.

```matlab
figure('Color',[1,1,1])
tic, sldemo_sweeptable_thd(24, 256), toc
```

Elapsed time is 1.201854 seconds.

**Total Harmonic Distortion for 24-bit 256 point sine wave synthesis table using Direct Look-Up and Linear Interpolation**

![Graph showing total harmonic distortion for 24-bit 256 point sine wave synthesis table using Direct Look-Up and Linear Interpolation.](image)

**Next Steps**

To take this example further, try different table precision and element counts to see the effect of each. Using CORDIC approximation, try different numbers of iterations to see the effects on accuracy and computation time. You can investigate different implementation options for waveform synthesis algorithms using automatic code
generation available from the Simulink Coder and production code generation using Embedded Coder™. Embedded Target products offer direct connections to a variety of real-time processors and DSPs, including connection back to the Simulink diagram while the target is running in real-time. The Signal Processing Toolbox and DSP System Toolbox offer prepackaged capabilities for designing and implementing a wide variety of sample-based and frame-based signal processing systems with MATLAB and Simulink.

bdclose('sldemo_tonegen');
bdclose('sldemo_tonegen_fixpt')
Accurate Zero-Crossing Detection

This example shows how zero crossings work in Simulink®. In this model, three shifted sine waves are fed into an absolute value block and saturation block. At exactly $t = 5$, the output of the switch block changes from the absolute value to the saturation block. Zero crossings in Simulink will automatically detect exactly when the switch block changes its output, and the solver will step to the exact time that the event happens. This can be seen by examining the output in the scope.
Spiral Galaxy Formation Simulation Using MATLAB Function Blocks

This model was inspired by the classic paper "Galactic Bridges and Tails" (Toomre & Toomre 1972). The original paper explained how disc shaped galaxies could develop spiral arms. Two disc shape galaxies originally are far apart. They then fly by each other and almost collide. Once the galaxies are close enough, mutual gravitational forces cause spiral arms to form.

Except for the "PlotAll" visualization block's use of plotting routines, all MATLAB® function blocks in this model support code generation with Simulink® Coder™ and Embedded Coder™.

• **Requirements:** For this example, Simulink® generates code for the simulation in a Simulink project directory created in the current working directory (pwd). If you do not want to affect the current directory (or if you cannot generate files in this directory), you should change your working directory.

**Opening the Model and Running the Simulation**

Open the model and run the simulation by executing the code below. If you are using MATLAB® Help, select the code you want to execute and press F9 (or select code, right click on selection and select "Evaluate Selection"). The Simulink model will load, compile, and run.

```matlab
model = 'sldemo_eml_galaxy';
open_system(model);
sim(model);
```
Spiral Galaxy Formation
(based on Toomre & Toomre, 1972)
**Model Description**

This section describes the model in detail and explains the role of each block in the model. The green blocks initialize the simulation, the orange blocks are the core of the simulation, and the yellow block makes the galaxy animation view.

**Initial Conditions**

The model requires initial conditions for each galaxy. The initial conditions are: galaxy radius in parsecs (rp), galaxy mass in solar mass units (cm), galaxy position in parsecs (pos), and galaxy velocity in m/s (vel).

In the model, constant blocks specify the initial conditions. The initial conditions have been chosen such that the galaxies will nearly collide at some point in time.
"ConstructGalaxy" Blocks

The initial conditions are passed to the MATLAB function blocks Construct Galaxy 1 and Construct Galaxy 2. These MATLAB function blocks contain MATLAB code that builds the galaxy models.

In a typical galaxy, most of the mass is concentrated in its center as a super-massive black hole and/or star agglomeration. We model the galaxy as a disc with radius \( r \) with most of its mass concentrated in the inner circle of radius \( r/3 \). In addition to this super-massive nucleus, the "ConstructGalaxy" MATLAB function block creates 349 random stars with masses ranging from 4 to 24 solar masses. These stars are randomly positioned within distance \( r/3 \) and \( r \) from the center of the galaxy. The stars initially move in circular orbits around the galaxy core. Every object (star or galaxy core) has mass, position \((x, y, z)\), and velocity \((V_x, V_y, V_z)\).

"Matrix Concatenation" Block

This block joins information about both galaxies. At this point the model has 700 objects: 1 core for each galaxy and 349 stars around each core. These 700 objects interact according to Newtonian mechanics.

"Partition" Block

This MATLAB function block separates all 700 objects into two groups: heavy bodies and light bodies. The heavy bodies are the galaxy cores. The light bodies are the stars. Because the galaxy cores are much heavier than individual stars, the model will consider only the heavy-heavy and heavy-light interactions. We can ignore the light-light body interactions. This will save a lot of time since 698 out of 700 bodies in the model are light.

"ApplyGravity" Block

This MATLAB function block uses Newtonian mechanics to compute the velocities and positions of the bodies at each step. The "combine" block is also a MATLAB function block. It merges the data about heavy and light objects together.

"PlotAll" Block

This MATLAB function block plots the bodies in a figure and updates the position of each star at every step in the simulation.
**Closing the Model**

Close the model without saving any changes. Clear data generated by simulation/example.

```matlab
close_system(model, 0);
clear model sldemo_eml_galaxy_output;
```

**Note:**

- In this model, signal logging is turned on (observe that the 'GalaxyBodies' signal has a little blue signal logging antenna on it). The model saves output data in a `Dataset` object. Any other data is saved/modified in the model workspace to avoid cluttering the MATLAB workspace.

- To modify signal logging settings, right-click on the signal line and select "Signal Properties". In this example, the signal name is 'GalaxyBodies' and the "Log signal data" checkbox is checked.

- The logged data is saved to the MATLAB workspace as a `Dataset` object with the name 'sldemo_eml_galaxy_output'. Information on the 'GalaxyBodies' signal can be retrieved from this object by typing `sldemo_eml_galaxy_output.get('GalaxyBodies')` which returns a `Simulink.SimulationData.Signal` object. See more about working with these objects in the "Simulink.SimulationData.Signal" documentation.

**Comments About the Model**

Despite its simplicity, this model provides insight into how our own galaxy evolved. This example allows the user to maximally simplify the model and run a fast simulation. The user can easily modify this example by adding more galaxies.

**References**

Modulo-4 Counter Using Flip-Flops

This example shows how to use Flip-Flop blocks (found in the Simulink® Extras Library) to implement a Modulo-4 counter. The model takes the output of a Modulo-4 counter and generates a half clock cycle width pulse on every fourth clock pulses. Effectively, it produces a pulse whenever both outputs of the Modulo-4 counter are equal to 1.
Counters Using Conditionally Executed Subsystems

This model shows the contrast between enabled subsystems and triggered subsystems for the same control signal, through the use of counter circuits. After running the simulation, the scope shows three plots.

The first of these plots shows the control signal which is the output of the discrete pulse generator.

The second plot shows the output of the counter circuit in the enabled subsystem. The counter increments while the subsystem is enabled and then holds its output when the subsystem is disabled. The counter resets to zero when the subsystem is re-enabled because the enable port was configured to reset states when enabling.

The third plot shows the output of the counter circuit in the triggered subsystem. The counter increments every rising edge of the control signal.

The signals are shown in a Signal Viewer scope that does not require a scope block to be put into the block diagram. Instead, the "scope" icon notation on the ports show that a Signal Viewer is attached. Double click on the scope icon to make the Signal Viewer appear if it is hidden.
Counters Using Conditionally Executed Subsystems

Copyright 1990-2014 The MathWorks, Inc.
Friction Model with Hard Stops

This example shows how to model friction one way in Simulink®. The two integrators in the model calculate the velocity and position of the system, which is then used in the Friction Model to calculate the friction force.

Running the simulation shows the initial condition response on the Scope.

Copyright 1990-2014 The MathWorks, Inc.
State Events

This example shows how to handle state events. Run the simulation and see the phase plane plot, where the state x1 is along the X-axis and the state x2 is along the Y-axis.
Bang-Bang Control Using Temporal Logic

This example shows how to use Stateflow® to model a bang-bang temperature control system for a boiler. The boiler dynamics are modeled in Simulink®.

The Bang-Bang Controller Stateflow chart shows how to use functions and Subcharts within Stateflow.
Inverted Pendulum with Animation

This example shows how to model an inverted pendulum. The animation is created using MATLAB® Handle Graphics®. The animation block is a masked S-function. For more information, use the context menu to look under the Animation block's mask and open the S-function for editing.
Double Spring Mass System

This example shows how to model a double spring-mass-damper system with a periodically varying forcing function. Associated with the example is an animation function that will automatically open a figure window and display to it. In this system, the only sensor is attached to the mass on the left, and the actuator is attached to the mass on the left. State estimation and LQR control are used.
Double Spring Mass System
Tank Fill and Empty with Animation

This example shows how to model the dynamics of liquid in a tank. The associated animation provides a graphical display of the tank as it empties and refills, based on user-defined tank parameters. The tank empties at the start of the simulation and again part way through the simulation. When the simulation is stopped, a plot is generated showing the liquid height and the states of the two valves.
Simulating Systems with Variable Transport Delay Phenomena

This example shows two cases where you can use Simulink® to model variable transport delay phenomena.

**Vertical Wheel Displacement on a One-Dimensional Car**

![Diagram of a car with speed v(t)](image)

**Figure 1:** Illustration of a car with speed v(t).

A car is running along a road with speed v(t). A sensor is installed at the front wheel to measure the vertical displacement Hi(t) of the front wheel caused by the road profile. If the wheels and road never lose contact, then the vertical displacement of the rear wheel, Ho(t), can be seen as a variable transport delay of Hi(t), which is determined by the length L between the two wheels and the speed v(t).
Vertical displacement of the wheels of a one dimensional car

Copyright 1990-2013 The MathWorks, Inc.
Figure 2: Vertical displacement of the wheels.

Incompressible Flow Through a Fixed Length Pipe

Figure 3: Illustration of a fixed-length pipe.
An incompressible flow goes through a pipe of length $L$ with speed $v(t)$. At the inlet, the flow temperature is $T_i$. We can model the temperature at the outlet $T_o$ as a variable transport delay of $T_i$. At time $t=0$, the pipe is empty and until $t=2$, there is no flow at the outlet. Thus, the output temperature before $t=2$ is the initial output temperature.

Open this model:
Figure 4: Incompressible flow through a fixed-length pipe.
Modeling a Foucault Pendulum

This example shows how to model a Foucault pendulum. The Foucault pendulum was the brainchild of the French physicist Leon Foucault. It was intended to prove that Earth rotates around its axis. The oscillation plane of a Foucault pendulum rotates throughout the day as a result of axial rotation of the Earth. The plane of oscillation completes a whole circle in a time interval $T$, which depends on the geographical latitude.

Foucault's most famous pendulum was installed inside the Paris Pantheon. This was a 28kg metallic sphere attached to a 67 meter long wire. This example simulates a 67 meter long pendulum at the geographic latitude of Paris.

Simulink® Model

The simplest way to solve the Foucault pendulum problem in Simulink® is to build a model that solves the coupled differential equations for the system. This model is shown in Figure 1. The equations that describe the Foucault pendulum are given below. For details on the physics of the model and the derivation of these equations, see Analysis and Physics.

\[
\begin{align*}
\ddot{x} &= 2\Omega \sin \lambda \dot{y} - \frac{g}{L} x \\
\ddot{y} &= -2\Omega \sin \lambda \dot{x} - \frac{g}{L} y
\end{align*}
\]

$x, y$ = pendulum bob coordinates as seen by an observer on Earth

$\Omega$ = Earth’s angular speed of revolution about its axis (rad/sec)

$g$ = acceleration of gravity (m/sec$^2$)

$L$ = length of the pendulum string (m)

$\lambda$ = geographic latitude (rad)

Opening the Model

Type `sldemo_foucault` in MATLAB® Command Window to open this model. This model logs simulation data to the variable `sldemo_foucault_output`. Logged signals have a blue indicator. Read more about signal logging in Simulink Help.
Figure 1: The Foucault pendulum model

Initial Conditions

This model loads the constants and initial conditions from the sldemo_foucault_data.m file. The contents of this file are shown in Table 1 below. You can modify simulation parameters directly in MATLAB workspace. The initial amplitude of
the pendulum must be small compared to pendulum length, because the differential equations are valid only for small oscillations.

**Table 1:** Initial conditions

- $g = 9.83$; % acceleration of gravity (m/sec^2)
- $L = 67$; % pendulum length (m)
- initial $x = L/100$; % initial $x$ coordinate (m)
- initial $y = 0$; % initial $y$ coordinate (m)
- initial $xdot = 0$; % initial $x$ velocity (m/sec)
- initial $ydot = 0$; % initial $y$ velocity (m/sec)
- $Omega=2*\pi/86400$; % Earth's angular velocity of rotation about its axis (rad/sec)
- $lambda=49/180*\pi$; % latitude in (rad)

**Running the Simulation**

Press the "Play" button on the toolbar in the model window to run the simulation. The simulation will use a variable step stiff solver, ode23t. It will simulate a Foucault pendulum for 3600 seconds (you can change the simulation time). The model uses a default relative tolerance $\text{RelTol} = 1e-6$. 
Figure 2: Foucault pendulum simulation results (simulation time of 3600 sec)
**Results**

The simulation results are shown in Figure 2 above. The simulation calculates the pendulum x and y coordinates, and the x and y velocity components of the pendulum.

The pendulum oscillation plane completes a 360 degree sweep in more than 24 hours. The sweep period is a function of geographic latitude $\lambda$ (see derivation in Analysis and Physics).

**Figure 3:** The Animation block shows how much the pendulum oscillation plane rotates in an hour.

After you run the simulation, double click the animation block to animate the results.

- Note: The "Animate Results" portion of the example requires Signal Processing Toolbox™. Double-clicking on the animation block will cause an error if it is not
installed. All other parts of the example will function correctly without the Signal Processing Toolbox.

The sldemo_foucault_animate.m file plots the position of the pendulum bob at different points in time. You can clearly see how the pendulum oscillation plane rotates.

- Note: If you are running the simulation at a large relative tolerance, the result will be numerically unstable over a long period of time. Make sure that you are using a stiff variable-step solver. Read more about numerical instability of stiff problems and solver performance in the "Exploring Variable-Step Solvers Using a Stiff Model" example.

**Closing the Model**

Close the model. Clear generated data.

**Analysis and Physics**

This section analyzes the Foucault pendulum and describes the physics behind it. The pendulum can be modeled as a point mass suspended on a wire of length L. The pendulum is located at the geographical latitude λ. It is convenient to use the reference frames shown in Figure 4: the inertial frame I (relative to the center of the Earth), and the non-inertial frame N (relative to an observer on Earth's surface). The non-inertial frame accelerates as a result of rotation.
Figure 4: The Inertial and Non-Inertial Frames for the Problem

Point O is the origin of the non-inertial frame N. It is the point on the surface of the earth beneath the point of suspension of the pendulum. The non inertial frame is chosen such that the z-axis points away from the center of the Earth and is perpendicular to Earth's surface. The x-axis points south and the y-axis points west.

As mentioned in the introduction, the oscillation plane of a Foucault pendulum rotates. The oscillation plane completes a full rotation in time $T_{rot}$ given by the following formula, where $T_{day}$ is the duration of one day (i.e. the time it takes the Earth to revolve around its axis once).
The sine factor requires further discussion. It is often incorrectly assumed that the oscillation plane of the pendulum is fixed in the inertial frame relative to the center of the Earth. This is only true at the north and south poles. To eliminate this confusion, think about the point S (see Figure 4), where the pendulum is suspended. In the inertial frame I, point S moves on a circle. The pendulum bob is suspended on a wire of constant length. For simplicity ignore the air friction. In the inertial frame I, there are only two forces that act on the bob - the wire tension \( T \) and the gravitational force \( F_g \).

The vector \( \mathbf{r} \) gives the position of the pendulum bob, B (see Figure 4). Newton's second law states that the sum of all forces acting on a body equals the mass times the acceleration of the body.

\[
m \ddot{\mathbf{r}} = \mathbf{T} + F_g
\]

Throughout this proof, the dots denote time derivatives, arrows denote vectors, caps denote unitary vectors (\( i \), \( j \), and \( k \) along x, y, and z axes). A dot above the vector arrow indicated the time derivative of the vector. An arrow above the dot indicated the vector of the time derivative. See the difference between total acceleration and radial acceleration below.

Total Acceleration:

\[
\ddot{\mathbf{r}} = \frac{d^2 \mathbf{r}}{dt^2} = \frac{d^2}{dt^2} \left( x \hat{i} + y \hat{j} + z \hat{k} \right)
\]

Radial Acceleration:

\[
\ddot{\mathbf{r}} = \left( \frac{d^2 r}{dt^2} \right) = \ddot{x} \hat{i} + \ddot{y} \hat{j} + \ddot{z} \hat{k}
\]

The acceleration of gravity points towards the center of the earth (negative z-direction).

\[
g = 9.83 \text{ m/}\text{sec}^2
\]

\[
\mathbf{g} = -g \hat{k}
\]

\[
m \ddot{\mathbf{r}} = \mathbf{T} - mg \hat{k}
\]
Decompose the acceleration term:

\[
r' = r_x \hat{i} + r_y \hat{j} + r_z \hat{k}
\]

\[
r'' = (\dot{r_x} \hat{i} + \dot{r_y} \hat{j} + \dot{r_z} \hat{k}) + r_x \ddot{\hat{i}} + r_y \ddot{\hat{j}} + r_z \ddot{\hat{k}}
\]

The time derivatives of unit vectors appear because the non-inertial reference frame N is rotating in space. This means that the unitary vectors \( \hat{i}, \hat{j}, \) and \( \hat{k} \) rotate in space. Their time derivatives are given below. Omega is Earth's angular velocity of revolution around its axis. The scalar Omega is the value of the angular velocity. The vectorial Omega is the vector angular velocity. Its direction is determined by the right hand rule.

\[
\dot{\hat{i}} = \Omega \times \hat{i}
\]

\[
\dot{\hat{j}} = \Omega \times \hat{j}
\]

\[
\dot{\hat{k}} = \Omega \times \hat{k}
\]

Rewrite the time derivative of the vector \( r \) relative to Omega.

\[
r'' = (\dot{r_x} \hat{i} + \dot{r_y} \hat{j} + \dot{r_z} \hat{k}) + \Omega \times r' = r'' + \Omega \times r'
\]

Similarly, express the second time derivative of the vector \( r \).

\[
r'' = r'' + 2\Omega \times r' + \Omega \times (\Omega \times r')
\]

\( r'' = \) acceleration in the non-inertial frame N (x, y, z components)

\( 2\Omega \times r' = \) Coriolis acceleration

\( \Omega \times (\Omega \times r') = \) additional term due to rotation of non-inertial frame N

To simplify this equation, assume that Omega for Earth is very small. This allows us to ignore the third term in the equation above. In fact, the second term (which is already much smaller than the first term) is four orders of magnitude greater than the third term. This reduces the equation to the following form:
Newton's Second Law can be written and decomposed into x, y, and z components as follows:

\[ m \ddot{r} = T - mg \hat{k} - 2m \Omega \times \hat{r} \]

\[ m \ddot{x} = T_x + 2m \Omega \dot{y} \sin \lambda \]

\[ m \ddot{y} = T_y - 2m \Omega (\dot{x} \sin \lambda + \dot{z} \cos \lambda) \]

\[ m \ddot{z} = T_z - mg + 2m \Omega \dot{y} \cos \lambda \]

The angular amplitude of oscillations is small. Therefore, we can ignore the vertical velocity and vertical acceleration (z-dot and z-double-dot). The string tension components can be expressed using small angle approximations, which also considerably simplify the problem, making it two-dimensional (see below).

\[ T_z = mg - 2m \Omega \dot{y} \cos \lambda \approx mg \]

\[ T_x = -T \frac{x}{L} \]

\[ T_y = -T \frac{y}{L} \]

\[ T_z = T \frac{L - z}{L} \approx T \]

**Characteristic Differential Equations**

Finally the physics of the problem can be described by the system of coupled equations given below. The x and y coordinates specify the position of the pendulum bob as seen by an observer on Earth.

\[ \ddot{x} - 2\Omega \sin \lambda \dot{y} + \frac{g}{L} x = 0 \]

\[ \ddot{y} + 2\Omega \sin \lambda \dot{x} + \frac{g}{L} y = 0 \]
Analytic Solution (Approximate)

The following is an analytic solution to the Foucault pendulum problem. Unfortunately, it is not exact. If you try to substitute the analytic solution into the differential equations, uncanceled terms of the order Omega squared will remain. However, because Omega is very small, we can ignore the uncanceled terms for practical purposes.

\[ \eta = x + i \cdot y \text{ (complex number)} \]

\[ i \dot{\eta} + (2i\Omega \sin \lambda) \dot{\eta} + \frac{g}{L} \eta = 0 \]

\[ \eta = \left( C_1 e^{i\sqrt{g/L}t} + C_2 e^{-i\sqrt{g/L}t} \right) e^{-i\Omega t \sin \lambda} \]

C₁, C₂ are complex integration constants

Actual Differential Equation System Is Asymmetric

During the derivation, terms involving Omega squared were ignored. This resulted in xy symmetry in the differential equations. If the Omega squared terms are taken into consideration, the differential equation system becomes asymmetric (see below).

\[ \ddot{x} - 2\Omega \sin \lambda \dot{y} + \left( \frac{g}{L} - \Omega^2 \sin^2 \lambda \right) x = 0 \]

\[ \ddot{y} + 2\Omega \sin \lambda \dot{x} + \left( \frac{g}{L} - \Omega^2 \right) y = 0 \]

You can easily modify the current Foucault pendulum model to account for asymmetric differential equations. Simply edit the corresponding Gain blocks that contain g/L and add the necessary expression. This change will introduce a very small overall correction to the numerical result.
Foucault Pendulum Model with VRML Visualization

This example shows how to solve the differential equations for the Foucault Pendulum problem and displays the pendulum bob movement in the VRML scene. You can modify the Pendulum location by changing the Latitude / Longitude constant values in the model and other parameters (g, Omega, L and initial conditions) in MATLAB® workspace.

There are two VRML scenes associated with this model. The first scene shows the Foucault pendulum itself, the second shows geographical position of the pendulum on the Earth. During the simulation, two arrow markers indicate the current and the initial position of the pendulum in the Earth coordinate system.

This model solves the differential equations for the Foucault Pendulum problem and displays the pendulum bob movement in the VRML scene. You can modify the Pendulum location by changing the Latitude / Longitude constant values in the model and other parameters (g, Ω, L and initial conditions) in MATLAB(R) workspace.
Foucault Pendulum Model with VRML Visualization
Exploring Variable-Step Solvers Using a Stiff Model

This example shows the behaviour of variable-step solvers in a Foucault pendulum model. Simulink® solvers ode45, ode15s, ode23, and ode23t are used as test cases. Stiff differential equations are used to solve this problem. There is no exact definition of stiffness for equations. Some numerical methods are unstable when used to solve stiff equations and very small step-sizes are required to obtain a numerically stable solution to a stiff problem. A stiff problem may have a fast changing component and a slow changing component.

Foucault pendulum is an example of a stiff problem. The pendulum completes an oscillation in a few seconds (fast changing component) whereas the Earth completes a rotation about its axis in a day (slow changing component). The oscillation plane of the pendulum slowly rotates because of Earth's axial rotation. Read more about the physics of a Foucault pendulum in Modelling a Foucault Pendulum.

The simulation calculates the position of the pendulum bob in the x-y plane as viewed by an observer on the surface of Earth. The total energy of the pendulum is then calculated and used to assess the performance of various Simulink solvers.

**Foucault Pendulum Model**

The Foucault pendulum is described by the system of coupled differential equations given below. Friction and air drag are not taken into consideration (this greatly simplifies the equations). A full derivation of these equations is given in the Foucault Pendulum example.

\[
\begin{align*}
\ddot{x} &= 2\Omega \sin \lambda \dot{y} - \frac{g}{L} x \\
\ddot{y} &= -2\Omega \sin \lambda \dot{x} - \frac{g}{L} y \\
\Omega &= \text{Earth's angular velocity of rotation around its axis} \\
\lambda &= \text{geographic latitude} \\
g &= \text{acceleration of gravity} \\
x \text{ and } y &= \text{coordinates of the pendulum bob}
\end{align*}
\]
The model shown in Figure 1 is used to solve the differential equations describing a Foucault pendulum. Type sldemo_solvers in MATLAB® Command Window to open the model. The example simulates a Foucault pendulum for 86400 seconds. The constants and initial conditions are saved in the model workspace.

Figure 1: Foucault pendulum model used to assess solver performance
Variable Step Solvers

This example investigates the performance of `ode45`, `ode15s`, `ode23`, and `ode23t` variable-step solvers. To read more about a particular solver, for example `ode45`, type `help ode45` in MATLAB Command Window.

<table>
<thead>
<tr>
<th>Solver</th>
<th>Problem Type</th>
<th>Order of Accuracy</th>
<th>When to Use</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ode45</code></td>
<td>Nonstiff</td>
<td>Medium</td>
<td>Most of the time. This should be the first solver you try.</td>
</tr>
<tr>
<td><code>ode23</code></td>
<td>Nonstiff</td>
<td>Low</td>
<td>For problems with crude error tolerances or for solving moderately stiff problems.</td>
</tr>
<tr>
<td><code>ode113</code></td>
<td>Nonstiff</td>
<td>Low to high</td>
<td>For problems with stringent error tolerances or for solving computationally intensive problems.</td>
</tr>
<tr>
<td><code>ode15s</code></td>
<td>Stiff</td>
<td>Low to medium</td>
<td>If <code>ode45</code> is slow because the problem is stiff.</td>
</tr>
<tr>
<td><code>ode23s</code></td>
<td>Stiff</td>
<td>Low</td>
<td>If using crude error tolerances to solve stiff systems and the mass matrix is constant.</td>
</tr>
<tr>
<td><code>ode23t</code></td>
<td>Moderately Stiff</td>
<td>Low</td>
<td>For moderately stiff problems if you need a solution without numerical damping.</td>
</tr>
<tr>
<td><code>ode23tb</code></td>
<td>Stiff</td>
<td>Low</td>
<td>If using crude error tolerances to solve stiff systems.</td>
</tr>
</tbody>
</table>

**Figure 2:** List of variable-step solvers available in Simulink

Assessing Solver Performance

There are different ways to assess the performance of a solver. If a problem has a closed form solution, you could compare the solver results with the expected theoretical result. You could also monitor the time it takes to simulate a model using a particular solver.

Unfortunately there is no exact theoretical solution for the Foucault Pendulum problem. There is an approximate closed form solution, however it cannot be used to assess solver performance (see the Foucault pendulum example).

Total Energy Conservation

This example assesses solver performance by verifying the energy conservation law. In a frictionless environment, the total energy of the pendulum must remain constant. The calculated energy of the pendulum, however, will not remain constant as a result of limited numeric accuracy.

This example calculates the normalized total energy of the pendulum at every time-step. The relative error in energy equals the change in total energy over the course of the simulation. Ideally, the relative error in energy must be zero because energy is conserved. Total energy is the sum of potential and kinetic energy. "NormalizeEnergy" block calculates normalized pendulum energy using the expressions given below.
Figure 3: Normalized energy versus time

Figure 3 shows a plot of normalized energy versus time as calculated using ode23 and ode23t. It is clear that in this particular problem ode23t is much more accurate than ode23. In the simulation that used ode23, the normalized pendulum energy decreased by more than 60%. A pendulum with lower energy has a lower oscillation amplitude. You can see this effect in Figure 4, where the amplitude of the pendulum calculated by ode23 decreases as the oscillation plane rotates.
Figure 4: Pendulum position as calculated using ode23 and ode23t

Figure 4 illustrates the difference between a stiff and a non-stiff solver. Each plot shows the position of the pendulum bob throughout a day (every 15th data point is plotted as a blue point). The black dot marks the initial position of the pendulum bob and the black line marks the initial pendulum oscillation plane. The red line indicates the oscillation plane after a day. The yellow line shows the oscillation plane at some intermediate point in time. Note that the oscillation plane of the pendulum does not complete a full rotation within a day. How fast the oscillation plane rotates depends on the geographical latitude (see details in the Foucault pendulum example). Observe that the pendulum amplitude in the left plot decreases whereas the amplitude in the right plot remains constant. For the same relative tolerance, RelTol=1e-5, the stiff solver gives a more accurate result, but requires more execution time.

Figure 5 shows a more detailed performance study of Simulink solvers. Four solvers were chosen to illustrate how relative error and simulation execution time vary as a function of relative tolerance. You can use the sldemo_solvers_mcode.m script for more extensive solver tests. This script generates C code from the model to speed up the simulations. Note that the script can take several minutes to execute.
Figure 5: Relative error and execution time for various solver settings. $\text{Abs}\text{Tol} = '1e-7' \text{ for all simulations.}$

Observe that in this example relative error does not decrease very significantly for relative tolerances below $1e-6$. This is a numeric solver limitation that depends on the particular model. Reducing the solver relative tolerance does not necessarily improve
accuracy. You need to estimate the minimal accuracy that is required for your problem and choose the solver accordingly to minimize simulation costs. For example you may want to know the position of the Foucault pendulum bob within a few centimeters. It is unnecessary to calculate the position of a pendulum within a few microns because you cannot measure the position accurately.

Conclusions

The numeric results of a simulation can behave differently depending on solver settings. This is crucial in the case of stiff problems. When working with stiff models, choose a solver that will give an accurate result at a minimal price. The relative tolerance (variable-step solvers) or step size (fixed-step solvers) have to be small enough that the result is numerically stable. Although some solvers are more efficient than others, stiff solvers are better suited for solving stiff problems. Variable step solvers are more robust than fixed-step solvers.
Exploring the Solver Jacobian Structure of a Model

The example shows how to use Simulink® to explore the solver Jacobian sparsity pattern, and the connection between the solver Jacobian sparsity pattern and the dependency between components of a physical system. A Simulink model that models the synchronization of three metronomes placed on a free moving base are used.

The Solver Jacobian Pattern

In general, the continuous part of a Simulink model can be written as:

\[
\begin{align*}
\dot{x} &= f(x, u) \\
y &= g(x, u)
\end{align*}
\]

where \(x\) are the continuous states and \(u\) are the inputs. \(y\) are the outputs.

The matrix:

\[ J_x = \frac{\partial f}{\partial x} \]

is called the system solver Jacobian matrix. When an implicit ODE solver is used to solve the system equations, \(J_x\) needs to be computed when needed. For example, the Jacobian matrix of a set of first order ODEs

\[
\begin{align*}
\dot{x}_1 &= x_1 x_2 \\
\dot{x}_2 &= x_2
\end{align*}
\]

is

\[
J_x = \begin{bmatrix}
\frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} \\
\frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2}
\end{bmatrix} = \begin{bmatrix}
x_2 & x_1 \\
0 & 1
\end{bmatrix}
\]

You can convert the solver Jacobian matrix to a Boolean sparse matrix by representing each non-zero element as 1, and each element that is always zero (hard zero) as 0. For example, the Boolean matrix corresponding to above Jacobian \(J_x\) is:

\[
J_{xp} = \begin{bmatrix}
1 & 1 \\
0 & 1
\end{bmatrix},
\]
where $J_{xp}$ is called the solver Jacobian pattern matrix.

The solver Jacobian pattern matrix can be generated directly from the original system equations by using the following rule:

$$J_{xp}(i, j) = \begin{cases} 
1 & \text{if } x_j \text{ is explicitly used to compute } \dot{x}_i \\
0 & \text{otherwise}
\end{cases}$$

As you can see, the solver Jacobian pattern matrix actually represents the dependency between the state variables and their derivatives. If computing of $\dot{x}_i$ needs the value of $x_j$, then there exists a dependency $\dot{x}_i \leftarrow x_j$ and $J_{xp}(i, j) = 1$. These dependencies are determined by the physical nature of the system, and thus by studying the solver Jacobian matrix, you can explore the physical structure of the physical system represented by the model. Simulink provides APIs for the user to get the solver Jacobian pattern matrix. The following shows how to access the solver Jacobian pattern and to use it to the study the model.

**The Pattern and Dependency: Synchronization of Metronomes**

Synchronization is defined as an adjustment of rhythms of oscillating objects due to their weak interaction [1]. One of the first scientifically documented observations of synchronization was reported by the Dutch scientist Christiaan Huygens, the inventor of pendulum clock [2]. Huygens discovered that two pendulums attached to the same beam supported by chairs would swing in exact opposite direction after some time. A similar set up used in this example is shown in Figure 1.
Figure 1: Set up used in this example: three metronomes on a moving base

Modeling the System

The model of this physical system can be divided into two parts:

- The metronomes mechanism
- The moving base

The metronomes mechanism

Referring to Figure 1, the dynamic equations of a single metronome on a moving base can be derived as[3]:

\[
\ddot{\theta} + \frac{mrg}{J} \sin \theta + \frac{b}{J} \left[ \left( \frac{\theta}{\theta_r} \right)^2 - 1 \right] \dot{\theta} + \left( \frac{rm \cos \theta}{J} \right) \ddot{x} = 0 \quad (eq.1)
\]

The first two terms describe a simple pendulum without friction. The third term is used to approximate the escapement* and any damping of the metronome. This term increases...
the angular velocity for $\theta < \theta_0$ and decreases it for $\theta > \theta_0$ [3]. The last term is the coupled effect from the moving base, in terms of an inertial force.

**The moving base**

Assuming the motion of the base is frictionless, then the center of mass of the system will not change and you can be derive the following:

$$x = -\frac{m}{M + 3m} r (\sin \theta_1 + \sin \theta_2 + \sin \theta_3),$$

where $M$ is the mass of the base and $m$ is the mass of the pendulum.

Then eq.1 can be changed to:

$$\ddot{x} = -\frac{m}{M + 3m} r \left[ \dot{\theta}_1 \cos \theta_1 - \sin \theta_1 (\dot{\theta}_1)^2 + \dot{\theta}_2 \cos \theta_2 - \sin \theta_2 (\dot{\theta}_2)^2 + \dot{\theta}_3 \cos \theta_3 - \sin \theta_3 (\dot{\theta}_3)^2 \right] \quad (eq.2)$$

The following shows the above system implemented using Simulink. The model contains three metronome subsystems and the moving base.
Simulation of this system shows an interesting phenomenon: Synchronization. It shows that all three metronomes with different initial phase angle eventually become synchronized with each other. Figure 3 shows the simulation results. The main cause for synchronization is the moving base that links all these metronomes together. This physical connection can be observed from the dynamic equation of each metronome.
Also, this physical connection can also be observed from the solver Jacobian pattern of this model. The following MATLAB® code shows how to get the solver Jacobian pattern.
Figure 3: The synchronized metronomes

Steps to Get the Solver Jacobian Pattern

% 1. Set the solver to be any implicit solver
   
   set_param(mdl, 'Solver', 'ode15s');
% 2. Set the solver Jacobian method to be Sparse perturbation *
    set_param(mdl, 'SolverJacobianMethodControl', 'SparsePerturbation');
% 3. Get the solver Jacobian object.
    J = Simulink.Solver.getSlvrJacobianPattern(mdl);
    disp('J = ');
    disp(J);
% 4. Show the pattern in a figure. use the method J.show
    J.show;
% 5. Explore the pattern with the given state name and other information
    stateNames = J.stateNames;
    disp('stateNames = ');
    disp(stateNames);

The results you will see are:

J =
    SlvrJacobianPattern with properties:
        Jpattern: [8x8 double]
        numColGroup: 6
        colGroup: [8x1 double]
        stateNames: {8x1 cell}
        blockHandles: [8x1 double]

stateNames =
    'sldemo_metro/Moving base/Integrator1'
    '(sldemo_metro/Metronome1/Integrator2).Theta1'
    '(sldemo_metro/Metronome2/Integrator2).Theta2'
    '(sldemo_metro/Metronome3/Integrator2).Theta3'
    '(sldemo_metro/Metronome3/Integrator1).Thetadot_3'
    '(sldemo_metro/Metronome2/Integrator1).Thetadot_2'
    '(sldemo_metro/Metronome1/Integrator1).Thetadot_1'
    'sldemo_metro/Moving base/Integrator'
Properties of the Solver Jacobian Pattern Object

The solver Jacobian pattern $J$ is an object. It contains the following properties:

- $Jpattern$: A sparse mxArray which is the Jacobian pattern.
- $numColGroup$: Number of column groups.
- $colgroup$: A column partition of the sparse pattern matrix.
- $stateNames$: A cell array containing the state name of each state.
- $blockHandles$: Block handles of the owner of each state.
Study of the Solver Jacobian Pattern

Referring to Figure 4. First, the solver Jacobian of this model is sparse and the number of non-zero element is 28. Secondly, the number of column groups is 6; is less than the number of states 8.

The row 1-4 indicates the following dependencies:

\[
\begin{align*}
\dot{x}_1 & \leftarrow x_8 (\text{baseposition} \leftarrow \text{baseposseed}) \\
\dot{x}_2 & \leftarrow x_7 (\theta_1 \leftarrow \text{Thetadot1}) \\
\dot{x}_3 & \leftarrow x_6 (\theta_2 \leftarrow \text{Thetadot2}) \\
\dot{x}_4 & \leftarrow x_5 (\theta_3 \leftarrow \text{Thetadot3})
\end{align*}
\]

These relations are clear since speed is the derivative of position. Row 5-8 shows the following relations:

\[
\begin{align*}
\dot{x}_5 & \leftarrow x_{2-7} \\
\dot{x}_6 & \leftarrow x_{2-7} \\
\dot{x}_7 & \leftarrow x_{2-7} \\
\dot{x}_8 & \leftarrow x_{2-7}
\end{align*}
\]

These relations show that to compute the angular acceleration of the metronomes or acceleration of the moving base, the angular position and angular speed of the metronomes are needed, but not the position and speed of the base. These relations can be found by studying eq.(1) and eq.(2) directly.

Conclusion

The Solver Jacobian pattern is a tool to study the data dependency relations between the derivatives of the state variables and the state variables. These relations usually reflect certain physical couplings in the physical system. By using the tools provided, you can discover these relations associated with a Simulink model, even without the original dynamic equations of the physical model.

References


- Escapement is a set of mechanism that drives the metronome. See [2] for for more details.
Double Bouncing Ball: Use of Adaptive Zero-Crossing Location

This example shows how to choose the correct zero-crossing location algorithm, based on the system dynamics. For Zeno dynamic systems, or systems with strong chattering, you can select the adaptive zero-crossing detection algorithm through the Configure pane:

---> Solver
    ---> Zero-crossing options
        ---> Algorithm: [Non-adaptive, Adaptive]

You can run this model by typing 'sldemo_doublebounce' at the MATLAB® command-line.

The Double Bouncing Ball System

The Simulink® model in this example is used to simulate two bouncing balls. They start from the ground with different initial speeds, and their ground levels will change at different times.

Open the model
Passing Zeno: Double Bouncing Ball

Copyright 1990-2013 The MathWorks, Inc.
**Double Bouncing Balls With Non-adaptive Zero-Crossing Location Algorithm**

If the Non-adaptive zero-crossing location algorithm is used, the consecutive zero-crossing error causes the simulation to stop. This system is actually a so-called 'Zeno dynamic system'. When either ball is very close to the ground, Simulink will hang because too many zero crossings are detected in a very short period.

**Figure 1:** The double bouncing ball model and animation
Double Bouncing Ball: Use of Adaptive Zero-Crossing Location

Passing Zeno: Double Bouncing Ball

Running Nonadaptive Algorithm...

Simulation Error

Nonadaptive
Adaptive
Stop
Simulink will stop the simulation of model 'sldemo_doublebounce' because the 1 zero crossing signal(s) identified below caused 10 consecutive zero crossing events in time interval between 14.271148216192614 and 14.27115007542968.

Number of consecutive zero-crossings : 10
Zero-crossing signal name : RelopInput
  Block type : RelationalOperator
  Block path : 'sldemo_doublebounce/Free Drop Ball 1/Compare'

----
Figure 2: Vertical displacement of both balls with Non-adaptive zero-crossing location algorithm.

The simulation does not complete and an error message is shown. The ground level changing events cannot be observed.
Double Bouncing Balls With Adaptive Zero-Crossing Location Algorithm

If the adaptive algorithm is selected, Simulink will adaptively turn on/off the process to precisely locate the zero-crossing time. The conditions to turn on/off the location are:

1) Zero-crossing signal value is below a threshold value. You can control the threshold value through the Configure pane:

   --> Solver
   --> Zero-crossing options
   --> Algorithm: [Adaptive]
   --> Signal threshold

2) Consecutive zero-crossing diagnostic is hit. You can define consecutive zero crossing through the Configure pane:

   --> Solver
   --> Solver diagnostic controls
   --> Time tolerance and
   --> Number of consecutive zero crossings.
Passing Zeno: Double Bouncing Ball

Simulation Complete.
(Default algorithm is Adaptive.)

Ground 1 level changed. Ground 2 level changed.
**Figure 3:** Vertical displacement of both balls with adaptive zero crossing location algorithm.
The simulation has completed. The ground level changing events can be observed. A warning is shown to inform you when searching for events is turned off.
Four Hydraulic Cylinder Simulation

This example shows how to use Simulink® to create a model with four hydraulic cylinders. See two related examples that use the same basic components: single cylinder model and model with two cylinders and load constraints.

- Note: This is a basic hydraulics example. You can more easily build hydraulic and automotive models using Simscape™ Driveline™ and Simscape Fluids™.

- **Simscape Fluids** provides component libraries for modeling and simulating fluid systems. It includes models of pumps, valves, actuators, pipelines, and heat exchangers. You can use these components to develop fluid power systems such as front-loader, power steering, and landing gear actuation systems. Engine cooling and fuel supply systems can also be developed with Simscape Fluids. You can integrate mechanical, electrical, thermal, and other systems using components available within the Simscape product family.

- **Simscape Driveline** provides component libraries for modeling and simulating one-dimensional mechanical systems. It includes models of rotational and translational components, such as worm gears, planetary gears, lead screws, and clutches. You can use these components to model the transmission of mechanical power in helicopter drivetrains, industrial machinery, vehicle powertrains, and other applications. Automotive components, such as engines, tires, transmissions, and torque converters, are also included.

**Modeling**

Figure 1 shows the top level diagram of the model. This model has a single pump and four actuators. The same pump pressure ($p_1$) drives each cylinder assembly and the sum of their flows loads the pump. Although each of the four control valves could be controlled independently, as in an active suspension system, in this case all four receive the same commands, a linear ramp in orifice area from zero to 0.002 sq.m.

**Opening the Model and Running the Simulation**

To open this model, type `sldemo_hydcyl4` at MATLAB® terminal (click on the hyperlink if you are using MATLAB Help). Press the "Play" button on the model toolbar to run the simulation.

- Note: The model logs relevant data to MATLAB workspace, into the Simulink.SimulationOutput object `out`. The signal logging data is stored within `out`, in
a structure called sldemo_hydcyl4_output. Logged signals have a blue indicator (see the model). Read more about Signal Logging in Simulink Help.
**Figure 1:** Four cylinder model and simulation results

**Model Description**

The pump flow begins at $0.005 \text{ m}^3/\text{sec}$ (just like in the single cylinder model), then it drops to $0.0025 \text{ m}^3/\text{sec}$ at $t=0.05 \text{ sec}$. The parameters $C_1, C_2, C_d, \rho$, and $V_{30}$ are identical to those in the single cylinder model. However, by assuming individual values for $K, A$, and $\beta$, each one of the four cylinders exhibit distinctive transient responses. The table below gives the characteristics of the four actuators.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Actuator1</th>
<th>Actuator2</th>
<th>Actuator3</th>
<th>Actuator4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spring Constant</td>
<td>$K$</td>
<td>$K/4$</td>
<td>$4K$</td>
<td>$K$</td>
</tr>
<tr>
<td>Piston Area</td>
<td>$Ac$</td>
<td>$Ac/4$</td>
<td>$4Ac$</td>
<td>$Ac$</td>
</tr>
<tr>
<td>Bulk Modulus</td>
<td>$\beta$</td>
<td>$\beta$</td>
<td>$\beta$</td>
<td>$\beta/1000$</td>
</tr>
</tbody>
</table>

$\beta = 7e8 \text{ Pa}$ [fluid bulk modulus]

$K = 5e4 \text{ N/m}$ [spring constant]

$Ac = 1e-3 \text{ m}^2$ [cylinder cross-sectional area]
The ratio of area and spring constant is the same for all pistons, so they should have the same steady state output. The dominant time constant for each actuator subsystem is proportional to

\[
\frac{A^2_c}{K}
\]

(result obtained from dimensional analysis), so we can expect the piston assembly 2 to be somewhat faster than assembly 1. The piston assembly 3 is expected to be slower than 1 or 2. The piston assembly 4 has a significantly lower bulk modulus beta (as would be the case with air), thus we expect piston 4 to respond more sluggishly than piston 1.

**Results**
The initial jolt of flow at $t=0$ is seen by the four actuators as a pressure impulse. The pump pressure ($p_1$), which is initially high, drops rapidly because there is a high flow demand from the four loads. During the initial transient (about 4 $\text{msec}$), distinct responses identify the individual dynamic characteristics of each assembly unit.

As predicted by the parameter values, actuator 2 responds much faster than actuator 1. The third and fourth pistons are much slower because they require more working fluid to move the same distance. In case 3, the piston displaces more volume due to its larger
cross-sectional area. In case 4, although the displaced volume is the same as in case 1, the device requires more fluid because it is subsequently compressed.

As the pump pressure falls to the level within the cylinders, the distinctions in behavior are blurred. The individual responses blend into an overall system response which maintains the flow balance between the components. At \( t = 0.05 \text{ sec} \), the pump flow drops to a level that is close to the equilibrium and the actuator flows are nearly zero. The individual steady state piston positions are equal, as predicted by the design.

**Closing the Model**

Close the model. Clear generated data.
Two Cylinder Model with Load Constraints

This example shows how to model a rigid rod supporting a large mass interconnecting two hydraulic actuators. The model eliminates the springs as it applies the piston forces directly to the load. These forces balance the gravitational force and result in both linear and rotational displacement.

See two related examples that use the same basic components: four cylinder model and single cylinder model.

- Note: This is a basic hydraulics example. You can more easily build hydraulic and automotive models using Simscape™ Driveline™ and Simscape Fluids™.

- Simscape Fluids provides component libraries for modeling and simulating fluid systems. It includes models of pumps, valves, actuators, pipelines, and heat exchangers. You can use these components to develop fluid power systems such as front-loader, power steering, and landing gear actuation systems. Engine cooling and fuel supply systems can also be developed with Simscape Fluids. You can integrate mechanical, electrical, thermal, and other systems using components available within the Simscape product family.

- Simscape Driveline provides component libraries for modeling and simulating one-dimensional mechanical systems. It includes models of rotational and translational components, such as worm gears, planetary gears, lead screws, and clutches. You can use these components to model the transmission of mechanical power in helicopter drivetrains, industrial machinery, vehicle powertrains, and other applications. Automotive components, such as engines, tires, transmissions, and torque converters, are also included.

Analysis and Physics of the Model

We assume the rotation angle of the rod is small. The equations of motion for the rod are given below in Equation Block 1. The equations describing the cylinder and pump behavior are the same as in the single cylinder example.

Equation Block 1:

\[
M \frac{d^2 z}{dt^2} = F_b + F_a + F_{ext}
\]

\[
I \frac{d^2 \theta}{dt^2} = \frac{L}{2} F_b - \frac{L}{2} F_a
\]
z – displacement at the center

\( M \) – total mass

\( F_a \) – piston A force

\( F_b \) – piston B force

\( F_{ext} \) – external force at center

\( \theta \) – clockwise angular displacement

\( I \) – moment of inertia of the rod

\( L \) – rod length

The positions and velocities of the individual pistons follow directly from the geometry. See the corresponding equations below in Equation Block 2.

**Equation Block 2:**

\[
\begin{align*}
  z_a &= z - \theta \frac{L}{2} \\
  z_b &= z + \theta \frac{L}{2} \\
  \frac{dz_a}{dt} &= \frac{dz}{dt} - \frac{d\theta}{dt} \frac{L}{2} \\
  \frac{dz_b}{dt} &= \frac{dz}{dt} + \frac{d\theta}{dt} \frac{L}{2} \\
  z_a &= \text{piston A displacement} \\
  z_b &= \text{piston B displacement}
\end{align*}
\]

**Opening the Model and Running the Simulation**

To open this model, type `sldemo_hydro` at MATLAB® terminal (click on the hyperlink if you are using MATLAB Help). Press the "Play" button on the model toolbar to run the simulation.
- Note: The model logs relevant data to MATLAB workspace into the Simulink.SimulationOutput object `out`. The signal logging data is stored within `out`, in a structure called `sldemo_hydrod_output`. Logged signals have a blue indicator (see the model). Read more about Signal Logging in Simulink Help.

- Note: The model logs all the continuous states of the system to MATLAB workspace in a structure called `xout`. Each state is assigned a name to facilitate data logging. The names of the states are available in the 'stateName' field of `xout.signals`. Read more about state names in Simulink Help.
Figure 1: Two cylinder model and simulation results

'Mechanical Load' Subsystem

This subsystem is shown in Figure 2. It solves the equations of motion, which we compute directly using standard Simulink blocks. It is assumed that the rotation angle is small. Look under the mask of the 'Mechanical Load' subsystem to see its structure (right click on the subsystem > select "Look Under Mask").
**Figure 2: 'Mechanical Load' subsystem**

**Simulation Parameters**

The parameters used in this simulation are identical to the parameters used in the single cylinder model, except for the following:

- \( L = 1.5 \text{ m} \)
- \( M = 2500 \text{ kg} \)
- \( I = 100 \text{ kg/m}^2 \)
- \( Q_{\text{max}} = 0.005 \text{ m}^3/\text{sec (constant)} \)
- \( C_2 = 3\times10^{-9} \text{ m}^3/\text{sec/Pa} \)
- \( F_{\text{ext}} = -9.81\times M \text{ Newtons} \)

Although the pump flow is constant, the model controls the valves independently. Initially, at \( t = 0 \), the cross-section of valve B is zero. It grows linearly to \( 1.2\times10^{-5} \text{ m}^2 \) at \( t = 0.01 \text{ sec} \), and then linearly decreases to zero at \( t = 0.02 \text{ sec} \). The cross-section of valve A is \( 1.2\times10^{-5} \text{ sq.m.} \) at \( t = 0 \) and it linearly decreases to zero at \( t = 0.01 \text{ sec} \), then it linearly increases to \( 1.2\times10^{-5} \text{ sq.m.} \) at \( t = 0.02 \text{ sec} \). Then the behavior of the valves A and B repeats periodically with the same pattern. In other words the valves A and B are 180 degrees out of phase.

**Results**

Figures 3 and 4 show the linear and angular displacements of the rod. The linear displacement response is typical of a type-one integrating system. The relative positions and the angular movement of the rod illustrate the response of the two pistons to the out-of-phase control signals (the cross-section of the valves A and B).
Figure 3: Linear displacement of the pistons and the load (load is in the middle of the rod)
**Figure 4:** Angular displacement of the rod

**Close Model**

Close the model and clear all generated data.

**Conclusions**

Simulink provides a productive environment for simulating hydraulic systems, offering enhancements that provide enormous productivity in modeling and flexibility in numerical methods. The use of masked subsystems and model libraries facilitates structured modeling with automatic component updates. As users modify library elements, the models that use the elements automatically incorporate the new versions. Simulink can use differential-algebraic equations (DAEs) to model some fluid elements as
incompressible and others as compliant, allowing efficient solutions for complex systems of interdependent circuits.

Models such as this one can ultimately be used as part of overall plant or vehicle systems. The hierarchical nature of Simulink allows independently developed hydraulic actuators to be placed, as appropriate, in larger system models (for example adding controls in the form of sensors or valves). In cases such as these, tools from the MATLAB Control System Toolbox™ can analyze and tune the overall closed-loop system. The MATLAB/Simulink environment can thus support the entire design, analysis, and modeling cycle.
Modeling a Fault-Tolerant Fuel Control System

This example shows how to combine Stateflow® with Simulink® to efficiently model hybrid systems. This type of modeling is particularly useful for systems that have numerous possible operational modes based on discrete events. Traditional signal flow is handled in Simulink while changes in control configuration are implemented in Stateflow. The model described below represents a fuel control system for a gasoline engine. The system is highly robust in that individual sensor failures are detected and the control system is dynamically reconfigured for uninterrupted operation.

Analysis and Physics

Physical and empirical relationships form the basis for the throttle and intake manifold dynamics of this model. The air-fuel ratio is computed by dividing the air mass flow rate (pumped from the intake manifold) by the fuel mass flow rate (injected at the valves). The ideal (i.e. stoichiometric) mixture ratio provides a good compromise between power, fuel economy, and emissions. The target air-fuel ratio for this system is 14.6. Typically, a sensor determines the amount of residual oxygen present in the exhaust gas (EGO). This gives a good indication of the mixture ratio and provides a feedback measurement for closed-loop control. If the sensor indicates a high oxygen level, the control law increases the fuel rate. When the sensor detects a fuel-rich mixture, corresponding to a very low level of residual oxygen, the controller decreases the fuel rate.

Modeling

Figure 1 shows the top level of the Simulink model. To open the model, type sldemo_fuelsys in MATLAB® Command Window. Press the Play button in the model window toolbar to run the simulation. The model loads necessary data into the model workspace from sldemo_fuelsys_data.m. The model logs relevant data to MATLAB workspace in a data structure called sldemo_fuelsys_output and streams the data to the Simulation Data Inspector. Logged signals are marked with a blue indicator while streaming signals are marked with the light blue badge (see Figure 1).

Note that loading initial conditions into the model workspace keeps simulation data isolated from data in other open models that you may have open. This also helps avoid MATLAB workspace cluttering. To view the contents of the model workspace select Modeling > Model Explorer, and click on Model Workspace from the Model Hierarchy list.

Notice that units are visible on the model and subsystem icons and signal lines. Units are specified on the ports and on the bus object. To learn more about units in Simulink see Simulink Units.
Figure 1: Top-level diagram for the fuel control system model

The Dashboard subsystem (shown in Figure 2) allows the user to interact with the model during simulation. The Fault Injection switches can be moved from the Normal to Fail position to simulate sensor failures, while the Engine Speed selector switch can be toggled to change the engine speed. The fuel and air/fuel ratio signals are visualized using the dashboard gauges and scopes to provide visual feedback during a simulation run.
The fuel_rate_control uses signals from the system's sensors to determine the fuel rate which gives a stoichiometric mixture. The fuel rate combines with the actual air flow in the engine gas dynamics model to determine the resulting mixture ratio as sensed at the exhaust.

The user can selectively disable each of the four sensors (throttle angle, speed, EGO and manifold absolute pressure [MAP]) by using the slider switches in the dashboard subsystem, to simulate failures. Simulink accomplishes this by binding slider switches to the value parameter of the constant block. Double-click on the dashboard subsystem to open the control dashboard to change the position of the switch. Similarly, the user can induce the failure condition of a high engine speed by toggling the engine speed switch on the dashboard subsystem. A Repeating Table block provides the throttle angle input and periodically repeats the sequence of data specified in the mask.

The fuel_rate_control block, shown in Figure 3, uses the sensor input and feedback signals to adjust the fuel rate to give a stoichiometric ratio. The model uses three subsystems to implement this strategy: control logic, airflow calculation, and fuel...
calculation. Under normal operation, the model estimates the airflow rate and multiplies the estimate by the reciprocal of the desired ratio to give the fuel rate. Feedback from the oxygen sensor provides a closed-loop adjustment of the rate estimation in order to maintain the ideal mixture ratio.

**Figure 3:** Fuel rate controller subsystem

**Control Logic**

A single Stateflow chart, consisting of a set of six parallel states, implements the control logic in its entirety. The four parallel states shown at the top of Figure 4 correspond to the four individual sensors. The remaining two parallel states at the bottom consider the status of the four sensors simultaneously and determine the overall system operating mode. The model synchronously calls the entire Stateflow diagram at a regular sample time interval of 0.01 sec. This permits the conditions for transitions to the correct mode to be tested on a timely basis.

To open the control_logic Stateflow chart, double click on it in the fuel_rate_control subsystem.
Figure 4: The control logic chart

When execution begins, all of the states start in their normal mode with the exception of the oxygen sensor (EGO). The O2_warmup state is entered initially until the warmup period is complete. The system detects throttle and pressure sensor failures when their measured values fall outside their nominal ranges. A manifold vacuum in the absence of a speed signal indicates a speed sensor failure. The oxygen sensor also has a nominal range for failure conditions but, because zero is both the minimum signal level and the bottom of the range, failure can be detected only when it exceeds the upper limit.

Regardless of which sensor fails, the model always generates the directed event broadcast Fail.INC. In this way the triggering of the universal sensor failure logic is independent of the sensor. The model also uses a corresponding sensor recovery event, Fail.DEC. The Fail state keeps track of the number of failed sensors. The counter
increments on each Fail.INC event and decrements on each Fail.DEC event. The model uses a superstate, Multi, to group all cases where more than one sensor has failed.

The bottom parallel state represents the fueling mode of the engine. If a single sensor fails, operation continues but the air/fuel mixture is richer to allow smoother running at the cost of higher emissions. If more than one sensor has failed, the engine shuts down as a safety measure, since the air/fuel ratio cannot be controlled reliably.

During the oxygen sensor warm-up, the model maintains the mixture at normal levels. If this is unsatisfactory, the user can change the design by moving the warm-up state to within the Rich_Mixture superstate. If a sensor failure occurs during the warm-up period, the Single_Failure state is entered after the warm-up time elapses. Otherwise, the Normal state is activated at this time.

A protective overspeed feature has been added to the model by creating a new state in the Fuel_Disabled superstate. Through the use of history junctions, we assured that the chart returns to the appropriate state when the model exits the overspeed state. As the safety requirements for the engine become better specified, we can add additional shutdown states to the Fuel_Disabled superstate.

**Sensor Correction**

When a sensor fails, an estimate of the sensor is computed. For example, open the pressure sensor calculation. Under normal sensor operation the value of the pressure sensor is used. Otherwise, the value is estimated.

```plaintext
ans =
    logical
    0
```
The estimate of manifold pressure is computed as a function of engine speed and throttle position. The value is computed conveniently using a Simulink function inside Stateflow.
Airflow Calculation

The Airflow Calculation block (shown in Figure 6) is the location for the central control laws. This block is found inside the fuel_rate_control subsystem (open this block). The block estimates the intake air flow to determine the fuel rate which gives the appropriate air/fuel ratio. Closed-loop control adjusts the estimation according to the residual oxygen feedback in order to maintain the mixture ratio precisely. Even when a sensor failure mandates open-loop operation, the most recent closed-loop adjustment is retained to best meet the control objectives.

Figure 6: Airflow estimation and correction

Equation 1

The engine's intake air flow can be formulated as the product of the engine speed, the manifold pressure and a time-varying scale factor.

\[ q = \frac{N}{4\pi} V_{cd} \nu \frac{P_m}{RT} = C_{pump}(N, P_m)NP_m = \text{intake mass flow} \]
\[ N = \text{engine angular speed (Rad/sec)} \]
\[ V_{col} = \text{engine cylinder displacement volume} \]
\[ \nu = \text{volumetric efficiency} \]
\[ P_m = \text{manifold pressure} \]
\[ R, T = \text{specific gas constant, gas temperature} \]

\( C_{pump} \) is computed by a lookup table and multiplied by the speed and pressure to form the initial flow estimate. During transients, the throttle rate, with the derivative approximated by a high-pass filter, corrects the air flow for filling dynamics. The control algorithm provides additional correction according to Equation 2.

**Equation 2**

\[ e_0 = 0.5 \text{ for } EGO \leq 0.5 \]
\[ e_0 = -0.5 \text{ for } EGO > 0.5 \]
\[ e_1 = K_i(N, P_m)e_0 \text{ for } EGO \leq 0.5 \]
\[ \dot{e}_2 = e_1 \text{ for LOW mode with valid EGO signal} \]
\[ \dot{e}_2 = 0 \text{ for RICH, DISABLE or EGO warmup} \]
\[ e_0, e_1, e_2 = \text{intermediate error signals} \]

**Figure 7:** Engine Gas Dynamics subsystem
The nonlinear oxygen sensor (EGO Sensor block) is found inside the Mixing & Combustion block (see Figure 8) within the Engine Gas Dynamics subsystem (see Figure 7). EGO Sensor is modeled as a hyperbolic tangent function, and it provides a meaningful signal when in the vicinity of 0.5 volt. The raw error in the feedback loop is thus detected with a switching threshold, as indicated in Equation 2. If the air-fuel ratio is low (the mixture is lean), the original air estimate is too small and needs to be increased. Conversely, when the oxygen sensor output is high, the air estimate is too large and needs to be decreased. Integral control is utilized so that the correction term achieves a level that brings about zero steady-state error in the mixture ratio.

The normal closed-loop operation mode, LOW, adjusts the integrator dynamically to minimize the error. The integration is performed in discrete time, with updates every 10 milliseconds. When operating open-loop however, in the RICH or O2 failure modes, the feedback error is ignored and the integrator is held. This gives the best correction based on the most recent valid feedback.

**Fuel Calculation**

The fuel_calc subsystem (within the fuel_rate_control subsystem, see Figure 9) sets the injector signal to match the given airflow calculation and fault status. The first input is the computed airflow estimation. This is multiplied with the target fuel/air ratio to get the commanded fuel rate. Normally the target is stoichiometric, i.e. equals the optimal air to fuel ratio of 14.6. When a sensor fault occurs, the Stateflow control logic sets the mode input to a value of 2 or 3 (RICH or DISABLED) so that the mixture is either slightly rich of stoichiometric or is shut down completely.
The fuel_calc subsystem (Figure 9) employs adjustable compensation (Figure 10) in order to achieve different purposes in different modes. In normal operation, phase lead compensation of the feedback correction signal adds to the closed-loop stability margin. In RICH mode and during EGO sensor failure (open loop), however, the composite fuel signal is low-pass filtered to attenuate noise introduced in the estimation process. The end result is a signal representing the fuel flow rate which, in an actual system, would be translated to injector pulse times.
Figure 10: Switchable compensation subsystem
Results and Conclusions

Simulation results are shown in Figure 11 and Figure 12. The simulation is run with a throttle input that ramps from 10 to 20 degrees over a period of two seconds, then goes back to 10 degrees over the next two seconds. This cycle repeats continuously while the engine is held at a constant speed so that the user can experiment with different fault conditions and failure modes. Click on a sensor fault switch in the dashboard subsystem to simulate the failure of the associated sensor. Repeat this operation to slide the switch back for normal operation.
Fault Tolerant Fuel Control System: Fuel Flow Rate

Baseline: All Sensors Functioning Properly

Throttle Sensor Failed

Speed Sensor Failed

EGO Sensor Failed

MAP Sensor Failed

Time (sec)
Figure 11: Comparing the fuel flow rate for different sensor failures

Figure 11 compares the fuel flow rate under fault-free conditions (baseline) with the rate applied in the presence of a single failure in each sensor individually. In each case note the nonlinear relationship between fuel flow and the triangular throttle command (shown in Figure 13). In the baseline case, the fuel rate is regulated tightly, exhibiting a small ripple due to the switching nature of the EGO sensor's input circuitry. In the other four cases the system operates open loop. The control strategy is proven effective in maintaining the correct fuel profile in the single-failure mode. In each of the fault conditions, the fuel rate is essentially 125% of the baseline flow, fulfilling the design objective of 80% rich.
Fault Tolerant Fuel Control System: Air Fuel Ratio

Baseline: All Sensors Functioning Properly

Throttle Sensor Failed

Speed Sensor Failed

EGO Sensor Failed

MAP Sensor Failed

Time (sec)
**Figure 12:** Comparing the air-fuel ratio for different sensor failures

Figure 12 plots the corresponding air/fuel ratio for each case. The baseline plot shows the effects of closed-loop operation. The mixture ratio is regulated very tightly to the stoichiometric objective of 14.6. The rich mixture ratio is shown in the bottom four plots of Figure 12. Although they are not tightly regulated, as in the closed-loop case, they approximate the objective of air/fuel (0.8*14.6=11.7).

![Throttle Command](image)

**Figure 13:** Throttle command

The transient behavior of the system is shown in Figure 14. With a constant 12 degree throttle angle and the system in steady-state, a throttle failure is introduced at \( t = 2 \) and corrected at \( t = 5 \). At the onset of the failure, the fuel rate increases immediately. The effects are seen at the exhaust as the rich ratio propagates through the system. The steady-state condition is then quickly recovered when closed-loop operation is restored.
Figure 14: Transient response to fault detection

Remarks

With animation enabled in the Stateflow debugger, the state transitions are highlighted in the Stateflow diagram (see Figure 4) as the various states are activated. The sequence of activation is indicated by changing colors. This closely coupled synergy between Stateflow and Simulink fosters the modeling and development of complete control systems. An engineer's concepts can develop in a natural and structured fashion with immediate visual feedback reinforcing each step.

Related Examples

Refer to other examples related to sldemo_fuelsys:

- Fixed-point design: fxpdemo_fuelsys
- Production C/C++ code generation: “Air-Fuel Ratio Control System with Stateflow Charts” (Simulink Coder)
• Fixed-point production C/C++ code generation: “Air-Fuel Ratio Control System with Fixed-Point Data” (Embedded Coder)
Using a Data Dictionary to Manage the Data for a Fuel Control System

This example shows how to use data dictionaries to manage the data for a fuel rate control system designed using Simulink® and Stateflow®. To familiarize yourself with the fuel rate control model see sldemo_fuelsys.

Familiarize Yourself with the Model

The sldemo_fuelsys_dd model is a closed-loop system containing a "plant" and "controller". The plant is used to validate the design of the controller. In this example, the plant and controller are represented by separate models that are referenced from the test harness model. Let’s take a look at these models.

Open and Compile the Test Harness Model

View the Engine Gas Dynamics System (Plant)

Double-click on the Engine Gas Dynamics block to open the plant model.
View the Fuel Rate Control System (Controller)

Double-click on the Fuel Rate Controller block to open the controller model.
Investigate the Data Used by the Controller

The global design data for the controller model is defined in a data dictionary. Using data dictionaries has many advantages over defining data in the base workspace. To learn about data dictionaries see the documentation for Simulink Data Dictionary.

The controller model is explicitly linked to a data dictionary. This link is set up on the Data tab of the Model Properties dialog.

The icon in the lower-left corner of the model window shows that this model is linked to a data dictionary. You can open the data dictionary in the Model Explorer by clicking on this icon.

This data dictionary contains parameter and signal objects that are used to configure the controller algorithm for simulation and code generation. It also contains a reference to another data dictionary that defines the data type objects used by this model.

Initially, this data dictionary is configured for a float-point controller, as is seen by the data type display on the signal lines in the controller model. You can easily switch between floating-point and fixed-point controllers by changing the types dictionary (sldemo_fuelsys_dd_types.sldd) to reference the appropriate dictionary:

- Floating-point types: sldemo_fuelsys_dd_float.sldd
- Fixed-point types: sldemo_fuelsys_dd_fixpt.sldd

Investigate the Units Used by the Components

Notice that units are visible on the model and subsystem icons and signal lines. Units are specified on the ports and on the bus, signal and parameter objects in the data dictionary. To learn more about units in Simulink see Simulink Units.

Simulate the Test Harness Model

The test harness model is also linked to a data dictionary (sldemo_fuelsys_dd.sldd). This data dictionary contains references to the data dictionaries for the plant and controller models but it does not contain any additional data.

Simulate the test harness model to validate the behavior of the controller in either the floating-point or fixed-point configuration.

Close the Example

Close the models and data dictionaries from this example.
Related Examples

Refer to other examples related to sldemo_fuelsys:

- Floating-point design: sldemo_fuelsys
- Fixed-point design: fxpdemo_fuelsys
- Production C/C++ code generation: “Air-Fuel Ratio Control System with Stateflow Charts” (Simulink Coder)
- Fixed-point production C/C++ code generation: “Air-Fuel Ratio Control System with Fixed-Point Data” (Embedded Coder)
Modeling Engine Timing Using Triggered Subsystems

This example shows how to model a four-cylinder spark ignition internal combustion engine from the throttle to the crankshaft output. We used well-defined physical principles supplemented, where appropriate, with empirical relationships that describe the system's dynamic behavior without introducing unnecessary complexity.

*Note: See the closed-loop engine model (an enhanced version of this model).

**Analysis and Physics**

This example describes the concepts and details surrounding the creation of engine models with emphasis on important Simulink® modeling techniques. The basic model uses the enhanced capabilities of Simulink to capture time-based events with high fidelity. Within this simulation, a triggered subsystem models the transfer of the air-fuel mixture from the intake manifold to the cylinders via discrete valve events. This takes place concurrently with the continuous-time processes of intake flow, torque generation and acceleration. A second model adds an additional triggered subsystem that provides closed-loop engine speed control via a throttle actuator. These models can be used as standalone engine simulations. Or, they can be used within a larger system model, such as an integrated vehicle and powertrain simulation, in the development of a traction control system.

This model is based on published results by Crossley and Cook (1991). It describes the simulation of a four-cylinder spark ignition internal combustion engine. The Crossley and Cook work also shows how a simulation based on this model was validated against dynamometer test data. The ensuing sections (listed below) analyze the key elements of the engine model that were identified by Crossley and Cook:

1. Throttle
2. Intake manifold
3. Mass flow rate
4. Compression stroke
5. Torque generation and acceleration

• Note: Additional components can be added to the model to provide greater accuracy in simulation and to more closely replicate the behavior of the system.
Throttle

The first element of the model is the throttle body. The control input is the angle of the throttle plate. The rate at which the model introduces air into the intake manifold can be expressed as the product of two functions:

1. an empirical function of the throttle plate angle only
2. a function of the atmospheric and manifold pressures

In cases of lower manifold pressure (greater vacuum), the flow rate through the throttle body is sonic and is only a function of the throttle angle. This model accounts for this low pressure behavior with a switching condition in the compressibility equations shown in Equation 1.

Equation 1

\[
\dot{m}_{\text{ai}} = f(\theta) \cdot g(P_m) = \text{mass flow rate into manifold (g/s)}
\]

\[
f(\theta) = 2.821 - 0.05231 \cdot \theta + 0.10299 \cdot \theta^2 - 0.00063 \cdot \theta^3
\]

\[
g(P_m) = 1; \text{ if } P_m \leq \frac{P_{\text{amb}}}{2}
\]

\[
g(P_m) = \frac{2}{P_{\text{amb}}} \sqrt{P_m P_{\text{amb}} - P_m^2}; \text{ if } \frac{P_{\text{amb}}}{2} \leq P_m \leq P_{\text{amb}}
\]

\[
g(P_m) = -\frac{2}{P_m} \sqrt{P_m P_{\text{amb}} - P_{\text{amb}}^2}; \text{ if } P_{\text{amb}} \leq P_m \leq 2P_{\text{amb}}
\]

\[
g(P_m) = -1; \text{ if } P_m \geq 2P_{\text{amb}}
\]

\[
\dot{m}_{\text{ai}} \rightarrow \text{mass flow rate into manifold (g/s)};
\]

\[
\theta \rightarrow \text{throttle angle (deg)};
\]

\[
P_m \rightarrow \text{manifold pressure (bar)};
\]

\[
P_{\text{amb}} \rightarrow \text{ambient (atmospheric) pressure (bar)};
\]
**Intake Manifold**

The simulation models the intake manifold as a differential equation for the manifold pressure. The difference in the incoming and outgoing mass flow rates represents the net rate of change of air mass with respect to time. This quantity, according to the ideal gas law, is proportional to the time derivative of the manifold pressure (see Equation 2). Note that, unlike the model of Crossley and Cook (see also references 3 through 5), this model doesn't incorporate exhaust gas recirculation (EGR), although this can easily be added.

**Equation 2**

\[
\hat{P}_m = \frac{RT}{V_m} (\dot{m}_{ai} - \dot{m}_{ao})
\]

- \( R \rightarrow \) specific gas constant;
- \( T \rightarrow \) temperature (K);
- \( V_m \rightarrow \) manifold volume (m\(^3\));
- \( \dot{m}_{ao} \rightarrow \) mass flow rate of air out of the manifold (g/s);
- \( \dot{P}_m \rightarrow \) rate of change of manifold pressure (bar/s);

**Intake Mass Flow Rate**

The mass flow rate of air that the model pumps into the cylinders from the manifold is described in Equation 3 by an empirically derived equation. This mass rate is a function of the manifold pressure and the engine speed.

**Equation 3**

\[
\dot{m}_{ao} = -0.366 + 0.08979 \cdot N \cdot P_m - 0.0337 \cdot N \cdot P_m^2 + 0.0001 \cdot N^2 \cdot P_m
\]

- \( N \rightarrow \) engine angular speed (rad/s);
- \( P_m \rightarrow \) manifold pressure (bar);

To determine the total air charge pumped into the cylinders, the simulation integrates the mass flow rate from the intake manifold and samples it at the end of each intake stroke.
event. This determines the total air mass that is present in each cylinder after the intake stroke and before compression.

**Compression Stroke**

In an inline four-cylinder four-stroke engine, 180 degrees of crankshaft revolution separate the ignition of each successive cylinder. This results in each cylinder firing on every other crank revolution. In this model, the intake, compression, combustion, and exhaust strokes occur simultaneously (at any given time, one cylinder is in each phase). To account for compression, the combustion of each intake charge is delayed by 180 degrees of crank rotation from the end of the intake stroke.

**Torque Generation and Acceleration**

The final element of the simulation describes the torque developed by the engine. An empirical relationship dependent upon the mass of the air charge, the air/fuel mixture ratio, the spark advance, and the engine speed is used for the torque computation (see Equation 4).

**Equation 4**

\[
Torque_{eng} = -181.3 + 379.36 \cdot m_a + 21.91 \cdot \left( \frac{A}{F} \right) - 0.85 \cdot \left( \frac{A}{F} \right)^2 + 0.26 \cdot \sigma - 0.0028 \cdot \sigma^2 + \\
0.027 \cdot N - 0.000107 \cdot N^2 + 0.00048 \cdot N \cdot \sigma + 2.55 \cdot \sigma \cdot m_a - 0.05 \cdot \sigma^2 \cdot m_a
\]

\( m_a \rightarrow \) mass of air in cylinder for combustion (g);

\( \left( \frac{A}{F} \right) \rightarrow \) air to fuel ratio;

\( \sigma \rightarrow \) spark advance (degrees before top - dead - center);

\( Torque_{eng} \rightarrow \) torque produced by the engine (Nm);

Calculate the engine angular acceleration using Equation 5

**Equation 5**

\[
J \dot{N} = Torque_{eng} - Torque_{load}
\]
\[ J \rightarrow \text{engine rotational moment of inertia } (kg \cdot m^2); \]

\[ \dot{N} \rightarrow \text{engine angular acceleration } (rad/s^2); \]

**Open-Loop Model**

We incorporated the model elements described above into an engine model using Simulink. The following sections describe the decisions we made for this implementation and the key Simulink elements used. This section shows how to implement a complex nonlinear engine model easily and quickly in Simulink environment. We developed this model in conjunction with Ken Butts, Ford Motor Company® (2).

Figure 1 shows the top level of the model. Note that, in general, the major blocks correspond to the high-level list of functions given in the model description in the preceding summary. Taking advantage of Simulink's hierarchical modeling capabilities, most of the blocks in Figure 1 are made up of smaller blocks. The following paragraphs describe these smaller blocks.

**Opening and Running the Simulation**

To open the model type `sldemo_engine` in MATLAB® terminal (click on the hyperlink if you are using MATLAB Help).

Press the "Play" button on the model toolbar to run the simulation.
Modeling Engine Timing Using Triggered Subsystems

Copyright 1990-2012 The MathWorks, Inc.
Figure 1: The top level of the engine model and simulation results

- Note: The model logs relevant data to MATLAB workspace in a structure called `sldemo_engine_output`. Logged signals have a blue indicator (see the model). Read more about Signal Logging in Simulink Help.

**Throttle/Manifold**

In the model, double click on the 'Throttle & Intake Manifold' subsystem to open it. It contains two other subsystems - the 'Throttle' and the 'Intake Manifold' subsystems. Open the 'Throttle' and 'Intake Manifold' to see their components.
Simulink models for the throttle and intake manifold subsystems are shown in Figure 2. The throttle valve behaves in a nonlinear manner and is modeled as a subsystem with three inputs. Simulink implements the individual equations, given in Equation 1, as function blocks. These provide a convenient way to describe a nonlinear equation of several variables. A 'Switch' block determines whether the flow is sonic by comparing the pressure ratio to its switch threshold, which is set at one half (Equation 1). In the sonic regime, the flow rate is a function of the throttle position only. The direction of flow is from the higher to lower pressure, as determined by the Sign block. With this in mind, the 'Min' block ensures that the pressure ratio is always unity or less.

The differential equation from Equation 2 models the intake manifold pressure. A Simulink function block computes the mass flow rate into the cylinder, a function of manifold pressure and engine speed (see Equation 3).

**Intake and Compression**

An integrator accumulates the cylinder mass air flow in the 'Intake' block (located inside the 'Throttle & Manifold' subsystem). The 'Valve Timing' block issues pulses that correspond to specific rotational positions in order to manage the intake and compression timing. Valve events occur each cam rotation, or every 180 degrees of crankshaft rotation. Each event triggers a single execution of the 'Compression' subsystem. The output of the trigger block within the 'Compression' subsystem then feeds back to reset the Intake.
integrator. In this way, although both triggers conceptually occur at the same instant in time, the integrator output is processed by the 'Compression' block immediately prior to being reset. Functionally, the 'Compression' subsystem uses a 'Unit Delay' block to insert 180 degrees (one event period) of delay between the intake and combustion of each air charge.

Consider a complete four-stroke cycle for one cylinder. During the intake stroke, the 'Intake' block integrates the mass flow rate from the manifold. After 180 degrees of crank rotation, the intake valve closes and the 'Unit Delay' block in the 'Compression' subsystem samples the integrator state. This value, the accumulated mass charge, is available at the output of the 'Compression' subsystem 180 degrees later for use in combustion. During the combustion stroke, the crank accelerates due to the generated torque. The final 180 degrees, the exhaust stroke, ends with a reset of the Intake integrator, prepared for the next complete 720 degrees cycle of this particular cylinder.

For four cylinders, we could use four 'Intake' blocks, four 'Compression' subsystems, etc., but each would be idle 75% of the time. We've made the implementation more efficient by performing the tasks of all four cylinders with one set of blocks. This is possible because, at the level of detail we've modeled, each function applies to only one cylinder at a time.

**Combustion**

Engine torque is a function of four variables. The model uses a 'Mux' block to combine these variables into a vector that provides input to the 'Torque Gen' block. A function block computes the engine torque (described empirically in Equation 4). The torque which loads the engine, computed by step functions in the Drag Torque block, is subtracted in the Engine Dynamics subsystem. The difference divided by the inertia yields the acceleration, which is integrated to arrive at the engine crankshaft speed.

**Plotting Simulation Results**

We used the following default inputs for the simulation:

\[
Throttle = 8.97 \text{ (deg)} \text{ if } t < 5
\]

\[
Throttle = 11.93 \text{ (deg)} \text{ if } t \geq 5
\]

\[
Load = 25 \text{ (Nm)} \text{ if } t \leq 2 \text{ or } t \geq 8
\]

\[
Load = 20 \text{ (Nm)} \text{ if } 2 < t \leq 8
\]
Try adjusting the throttle to compensate for the load torque. Figure 3 shows the simulated engine speed, the throttle commands which drive the simulation, and the load torque which disturbs it.

**Figure 3a**: Open-loop simulation inputs
Figure 3b: Open-loops simulation results

Closing Model

Close the model. Clear generated data.

Conclusions

The ability to model nonlinear, complex systems, such as the engine model described here, is one of Simulink's key features. The power of the simulation is evident in the presentation of the models above. Simulink retains model fidelity, including precisely timed cylinder intake events, which is critical in creating a model of this type. The basic engine model shows the flexibility of Simulink.
*Note: See the closed-loop engine model (an enhanced version of this model).

**References**


Engine Timing Model with Closed Loop Control

This example shows how to enhance a version of the open-loop engine model (sldemo_engine - described in "Modeling Engine Timing Using Triggered Subsystems" example). This model, sldemo_enginewc, contains a closed-loop and shows the flexibility and extensibility of Simulink® models. In this enhanced model, the objective of the controller is to regulate engine speed with a fast throttle actuator, such that changes in load torque have minimal effect. This is easily accomplished in Simulink by adding a discrete-time PI controller to the engine model.

Closed-Loop Model

We chose a control law which uses proportional plus integral (PI) control. The integrator is needed to adjust the steady-state throttle as the operating point changes, and the proportional term compensates for phase lag introduced by the integrator.

- Note: See the open-loop engine model (this model is an enhanced version of the open-loop model).

Equation 1

\[ \theta = K_p (N_{set} - N) + K_I \int (N_{set} - N) dt \]

\[ N_{set} = \text{speed set point (rpm)} \]

\[ K_p = \text{proportional gain} \]

\[ K_I = \text{integral gain} \]

Opening and Running the Simulation

To open this model type sldemo_enginewc at MATLAB® terminal (click on the hyperlink if you are using MATLAB Help). Press the "Play" button on the model toolbar to run the simulation.

- Note: The model logs relevant data to MATLAB workspace in a structure called sldemo_enginewc_output. Logged signals have a blue indicator (see the model). Read more about Signal Logging in Simulink Help.
**Figure 1:** Closed-loop engine model and simulation results

In this model we employ a discrete-time controller, which is suitable for microprocessor implementation. The integral term in Equation 1 must thus be realized with a discrete-time approximation. As is typical in the industry, the controller execution is synchronized with the engine's crankshaft rotation. The controller is embedded in a triggered subsystem that is triggered by the valve timing signal described above.

The detailed construction of the 'Controller' subsystem is illustrated in Figure 2. Of note is the use of the 'PID Controller' block. This block implements a proportional-integral control system in discrete time. Note the setting for sample time set (internally) at -1. This indicates that the block inherits its sample time, in this case executing each time the subsystem is triggered. The key component that makes this a triggered subsystem is the 'Trigger' block shown at the bottom of Figure 2. Any subsystem can be converted to a triggered subsystem by dragging a copy of this block into the subsystem diagram from the Simulink Connections library.
Results

Typical simulation results are shown in Figure 3. The speed set point steps from 2000 rpm to 3000 rpm at $t = 5 \text{ sec}$. The torque disturbances are identical to those used in sldemo_engine, the open-loop model (open the other engine model). Note the quick transient response, with zero steady-state error. Several alternative controller tunings ($K_i$ and $K_p$) are shown. These can be adjusted by the user at MATLAB command line. This allows the engineer to understand the relative effects of parameter variations.
Conclusions

The ability to model nonlinear, complex systems, such as the engine model described here, is one of Simulink's key features. The power of the simulation is evident in the presentation of the models above. Simulink retains model fidelity, including precisely timed cylinder intake events, which is critical in creating a model of this type. The complete speed control system shows the flexibility of Simulink. In particular, the
Simulink modeling approaches allow rapid prototyping of an interrupt-driven engine speed controller.

- Note: See the open-loop engine model (this model is an enhanced version of the open-loop model).

**References**


Building a Clutch Lock-Up Model

This example shows how to use Simulink® to model and simulate a rotating clutch system. Although modeling a clutch system is difficult because of topological changes in the system dynamics during lockup, this example shows how Simulink’s enabled subsystems easily handle such problems. We illustrate how to employ important Simulink modeling concepts in the creation of the clutch simulation. Designers can apply these concepts to many models with strong discontinuities and constraints that may change dynamically.

In the example, we use enabled subsystems to build the clutch model. Two enabled subsystems model the clutch dynamics in either the locked or unlocked position. After running the simulation, a GUI opens. Checking any of the boxes on the GUI produces a plot of any of the selected variables (versus time).

There is an alternative clutch model (sldemo_clutch_if) that uses 'If/Else' blocks instead of enabled subsystems (open the other clutch model).

Analysis and Physics

The clutch system in this example consists of two plates that transmit torque between the engine and transmission (see Figure 1). There are two distinct modes of operation:

1) slipping - the two plates have differing angular velocities

2) lockup - the two plates rotate together.

Handling the transition between these two modes presents a modeling challenge. As the system loses a degree of freedom upon lockup, the transmitted torque goes through a step discontinuity. The magnitude of the torque drops from the maximum value supported by the friction capacity to a value that is necessary to keep the two halves of the system spinning at the same rate. The reverse transition, break-apart, is likewise challenging, as the torque transmitted by the clutch plates exceeds the friction capacity.
Figure 1: The clutch system, analyzed using a lumped-parameter model

Variables Used

The following variables are used in the analysis and modeling.

\[ T_{in} \] = input (engine) torque;
\[ F_n \] = normal force between friction plates;
\[ I_e, I_v \] = moments of inertia for the engine and for the transmission/vehicle;
\[ b_e, b_v \] = damping rates at the engine and transmission/vehicle sides of the clutch;
\[ \mu_k, \mu_s \] = kinetic and static coefficients of friction;
\[ \omega_e, \omega_v \] = angular speeds of the engine and transmission input shafts;
\[ r_1, r_2 \] = inner and outer radii of the clutch plate friction surfaces;
\[ R \] = equivalent net radius;
\[ T_{cl} \] = torque transmitted through the clutch;
\[ T_f \] = friction torque required of the clutch to maintain lockup;
Equation 1

The state equations for the coupled system are derived as follows:

\[ I_c \ddot{\omega}_c = T_{in} - b_c \omega_c - T_{cl} \]
\[ I_v \ddot{\omega}_v = T_{cl} - b_v \omega_v \]

Equation 2

The torque capacity of the clutch is a function of its size, friction characteristics, and the normal force that is applied.

\[ (T_f)_{max} = \int \int_A \frac{r \times F_f}{A} \, da = \frac{F_n \mu}{\pi (r_2^2 - r_1^2)} \int_{r_1}^{r_2} \int_0^{2\pi} r^2 \, dr \, d\theta = \frac{2}{3} RF_n \mu \]
\[ R = \frac{r_2^3 - r_1^3}{r_2^2 - r_1^2} \]

Equation 3

When the clutch is slipping, the model uses the kinetic coefficient of friction and the full capacity is available, in the direction that opposes slip.

\[ T_{f_{max}} = \frac{2}{3} RF_n \mu_k \]
\[ T_{cl} = \text{sgn}(\omega_c - \omega_v) T_{f_{max}} \]

where sgn denotes the sign function.

Equation 4

When the clutch is locked, the angular velocities of the engine and transmission input shafts are the same, and the system torque acts on the combined inertia as a single unit. So, we combine the differential equations (Equation 1) into a single equation for the locked state.

\[ \omega_c = \omega_v = \omega \]
\[ (I_c + I_v) \dot{\omega} = T_{in} - (b_c + b_v) \omega \]
Equation 5

Solving Equation 1 and Equation 4, the torque transmitted by the clutch while locked is:

\[ T_{cl} = T_f = \frac{I_v T_{in} - (I_v b_e - I_e b_v) \omega}{I_v + I_e} \]

Equation 6

The clutch thus remains locked unless the magnitude of \( T_f \) exceeds the static friction capacity, \( T_{f_{max}} \).

\[ T_{f_{max}} = \frac{2}{3} RF_n \mu_s \]

The state diagram in Figure 2 describes the overall behavior of the clutch.

**Figure 2:** A state diagram describing the friction mode transitions

**Modeling**

There are two methods for solving this type of problem:

1) Compute the clutch torque transmitted at all times, and employ this value directly in the model

2) Use two different dynamic models and switch between them at the appropriate times
Because of its overall capabilities, Simulink can model either method. In this example, we describe a simulation for the second method. In the second method, switching between two dynamic models must be performed with care to ensure that the initialized states of the new model match the state values immediately prior to the switch. But, in either approach, Simulink facilitates accurate simulation due to its ability to recognize the precise moments at which transitions between lockup and slipping occur.

The simulation model for the clutch system uses enabled subsystems, a particularly useful feature in Simulink. The simulation can use one subsystem while the clutch is slipping and the other when it is locked. A diagram of the Simulink model appears in Figure 3.

- Opening the Model and Running the Simulation

To open this model type sldemo_clutch in MATLAB® terminal (click on the hyperlink if you are using MATLAB Help).

When the model is open, press the "Play" button on the model toolbar to run the simulation.

• Note: If you are using MATLAB Help, you can execute code from the example page by selecting the code and pressing F9. You can also Select Code > Right Click > Select "Evaluate Selection".
Building a Clutch Lock-Up Model
An Example of Enabled Subsystems

Copyright 1990-2017 The MathWorks, Inc.
Figure 3: Top level diagram for the clutch model

- **Note:** The model logs relevant data to MATLAB workspace in a structure called `sldemo_clutch_output`. Logged signals have a blue indicator (see the model). Read more about Signal Logging in Simulink Help.

**- The 'Unlocked' Subsystem**

Double click on the 'Unlocked' subsystem in the model window to open it. This subsystem models both sides of the clutch, coupled by the friction torque. It is constructed around the integrator blocks which calculate engine and vehicle speeds (see Figure 4). The model uses gain, multiplication, and summation blocks to compute the speed derivatives (acceleration) from the states and the subsystem inputs of engine torque, \( T_{in} \), and clutch capacity, \( T_{fmaxk} \).
Figure 4: The 'Unlocked' subsystem

Enabled subsystems, such as 'Unlocked', feature several other noteworthy characteristics. The 'Enable' block at the top of the diagram in Figure 4, defines the model as an enabled subsystem. To create an enabled subsystem, we group the blocks together like any other subsystem. We then insert an 'Enable' block from the Simulink Connections library. This means that:

- An enable input appears on the subsystem block, identified by the pulse-shaped symbol used on the 'Enable' block itself.
- The subsystem executes only when the signal at the enable input is greater than zero.

In this example, the 'Unlocked' subsystem executes only when the supervising system logic determines that it should be enabled.
There is another important consideration when using systems that can be enabled or disabled. When the system is enabled, the simulation must reinitialize the integrators to begin simulating from the correct point. In this case, both sides of the clutch are moving at the same velocity the moment it unlocks. The 'Unlocked' subsystem, which had been dormant, needs to initialize both integrators at that speed in order to keep the system speeds continuous.

The simulation uses 'From' blocks to communicate the state of the locked speed to the initial condition inputs of the two integrators. Each 'From' block represents an invisible connection between itself and a 'Goto' block somewhere else in the system. The 'Goto' blocks connect to the state ports of the integrators so that the model can use these states elsewhere in the system without explicitly drawing in the connecting lines.

- The 'Locked' Subsystem

Open the 'Locked' subsystem by double clicking on it in the model window. This is another enabled subsystem in the clutch model (see Figure 5). It uses a single state to represent the engine and vehicle speeds. It computes acceleration as a function of the speed and input torque. As in the 'Unlocked' case, a 'From' block provides the integrator initial conditions and a 'Goto' block broadcasts the state for use elsewhere in the model. While simulating, either the 'Locked' or the 'Unlocked' subsystem is active at all times. Whenever the control changes, the states are neatly handed off between the two.
Figure 5: The 'Locked' Subsystem

- The 'Friction Mode Logic' Subsystem

The 'Friction Mode Logic' subsystem (shown in Figure 6) computes the static and kinetic friction (with the appropriate friction coefficient) according to the following formula:

\[ T_{f_{\text{max}}} = \frac{2}{3} RF_n \mu \]

Open the 'Friction Mode Logic' subsystem by double clicking on it in the model window.

Figure 6: The 'Friction Mode Logic' Subsystem

- Other Components

The remaining blocks calculate the torque required for lockup (Equation 5), and implement the logic described in Figure 2. One key element is located in the 'Lockup Detection' subsystem within the 'Friction Mode Logic' subsystem. This is the 'Simulink Hit Crossing' block which precisely locates the instant at which the clutch slip reaches zero. This places the mode transition at exactly the right moment.
- **System Inputs**

The system inputs are normal force, $F_n$, and engine torque, $T_{in}$. Each of these is represented by a matrix table in the model workspace. The inputs are plotted in Figure 7. You can visualize various signals by checking the corresponding boxes on the 'Clutch Demo Signals' GUI.

![Normal Force (F_n) and Engine Torque (T_{in})](image)

**Figure 7**: System inputs: normal force and engine torque

- **Results**

The following parameter values are used to show the simulation. These are not meant to represent the physical quantities corresponding to an actual system, but rather to facilitate a meaningful baseline example.
For the inputs shown above, the system velocities behave as shown in Figure 8 below. The simulation begins in the Unlocked mode, with an initial engine speed flare as the vehicle side accelerates its larger inertia. At about \( t = 4 \text{ sec} \), the velocities come together and remain locked, indicating that the clutch capacity is sufficient to transmit the torque. At \( t = 5 \text{ sec} \), the engine torque begins to decrease, as does the normal force on the friction plates. Consequently, the onset of slip occurs at about \( t = 6.25 \text{ sec} \) as indicated by the separation of the engine and vehicle speeds.
Figure 8: Angular velocities of the engine, vehicle and shaft for default inputs

Notice that the various states remain constant while they are disabled. At the time instants at which transitions take place, the state hand-off is both continuous and smooth. This is a result of supplying each integrator with the appropriate initial conditions to use when the state is enabled.

Closing Model

Close the model. Clear generated data.
Conclusions

This example shows how to use Simulink and its standard block library to model, simulate, and analyze a system with topological discontinuities. This is a powerful example of the 'Hit Crossing' block and how it can be used to capture specific events during a simulation. The Simulink model of this clutch system can serve as a guide when creating models with similar characteristics. You can apply the principles used in this example to any system with topological discontinuities.
Modeling Clutch Lock-Up Using If Blocks

This example shows how to use If/Else subsystems to build a clutch model. An 'If' subsystem models the clutch dynamics in the locked position while an 'Else' subsystem models the unlocked position. One or the other is enabled using the 'If' block. The dot-dashed lines from the 'If' block denote control signals, which are used to enable If/Else (or other conditional) subsystems.

This is an alternative to the clutch model with enabled subsystems - sldemo_clutch (open the other clutch model).

Analysis and Physics

The clutch system in this example consists of two plates that transmit torque between the engine and transmission. There are two distinct modes of operation:

1) slipping - the two plates have differing angular velocities
2) lockup - the two plates rotate together.

Handling the transition between these two modes presents a modeling challenge. As the system loses a degree of freedom upon lockup, the transmitted torque goes through a step discontinuity. The magnitude of the torque drops from the maximum value supported by the friction capacity to a value that is necessary to keep the two halves of the system spinning at the same rate. The reverse transition, break-apart, is likewise challenging, as the torque transmitted by the clutch plates exceeds the friction capacity.

• Note: You can find a detailed analysis of this system, including equations and diagrams, in the example for the clutch model with enabled subsystems.

Modeling

You can use the following two methods for solving this problem:

1) Compute the clutch torque transmitted at all times, and employ this value directly in the model
2) Use two different dynamic models and switch between them at the appropriate times

Simulink® can model either method. In this example, we describe a simulation for the second method. Switching between two dynamic models must be performed with care to ensure that the initialized states of the new model match the state values immediately
prior to the switch. In either approach, Simulink facilitates accurate simulation due to its ability to recognize the precise moments at which transitions between lockup and slipping occur.

**Opening the Model and Running the Simulation**

To open this model type `sldemo_clutch_if` in MATLAB® terminal (click on the hyperlink if you are using MATLAB Help).

When the model is open, press the "Play" button on the model toolbar to run the simulation.
Figure 1: Top level diagram for the clutch model

- Note: The model logs relevant data to MATLAB workspace in a structure called `sldemo_clutch_if_output`. Logged signals have a blue indicator (see the model). Read more about Signal Logging in Simulink Help.

The 'If' Block

The 'If' block uses the `LockedFlag` signal to switch between the 'Locked' and 'Unlocked' subsystems. Double click on the 'If' block in the model to set its parameters (see Figure 2). `LockedFlag` represents the status of the clutch. `LockedFlag = 1` if the clutch is locked and `LockedFlag = 0` if the clutch is unlocked.
The other components of this model are similar to those used in the clutch model with enabled subsystems (open the other clutch model). See a detailed description in Building a Clutch Lock-Up Model example.
**Results**

The inputs for this model are the same as for the model that uses enabled subsystems (sldemo_clutch). System velocities behave as shown in Figure 3 below. As expected, the results obtained from sldemo_clutch and sldemo_clutch_if are identical.

**Figure 3:** Angular velocities of the engine, vehicle and shaft for default inputs

**Closing Model**

Close the model. Clear generated data.
Conclusions

This example shows how to use 'If' blocks in Simulink to model a system with topological discontinuities. This is an alternative to using enabled subsystems.
Modeling an Anti-Lock Braking System

This example shows how to model a simple model for an Anti-Lock Braking System (ABS). It simulates the dynamic behavior of a vehicle under hard braking conditions. The model represents a single wheel, which may be replicated a number of times to create a model for a multi-wheel vehicle.

This model uses the signal logging feature in Simulink®. The model logs signals to the MATLAB® workspace where you can analyze and view them. You can view the code in sldemo_absbrakeplots.m to see how this is done.

In this model, the wheel speed is calculated in a separate model named sldemo_wheelspeed_absbrake. This component is then referenced using a 'Model' block. Note that both the top model and the referenced model use a variable step solver, so Simulink will track zero-crossings in the referenced model.

Analysis and Physics

The wheel rotates with an initial angular speed that corresponds to the vehicle speed before the brakes are applied. We used separate integrators to compute wheel angular speed and vehicle speed. We use two speeds to calculate slip, which is determined by Equation 1. Note that we introduce vehicle speed expressed as an angular velocity (see below).

\[ \omega_v = \frac{V}{R} \]  (equals the wheel angular speed if there is no slip)

\textbf{Equation 1}

\[ \omega_v = \frac{V_v}{R_r} \]

\[ slip = 1 - \frac{\omega_w}{\omega_v} \]

\[ \omega_w = \text{vehicle speed divided by wheel radius} \]

\[ V_v = \text{vehicle linear velocity} \]

\[ R_r = \text{wheel radius} \]
\[ \omega_w = \text{wheel angular velocity} \]

From these expressions, we see that slip is zero when wheel speed and vehicle speed are equal, and slip equals one when the wheel is locked. A desirable slip value is 0.2, which means that the number of wheel revolutions equals 0.8 times the number of revolutions under non-braking conditions with the same vehicle velocity. This maximizes the adhesion between the tire and road and minimizes the stopping distance with the available friction.

**Modeling**

The friction coefficient between the tire and the road surface, \( \mu \), is an empirical function of slip, known as the mu-slip curve. We created mu-slip curves by passing MATLAB variables into the block diagram using a Simulink lookup table. The model multiplies the friction coefficient, \( \mu \), by the weight on the wheel, \( W \), to yield the frictional force, \( F_f \), acting on the circumference of the tire. \( F_f \) is divided by the vehicle mass to produce the vehicle deceleration, which the model integrates to obtain vehicle velocity.

In this model, we used an ideal anti-lock braking controller, that uses 'bang-bang' control based upon the error between actual slip and desired slip. We set the desired slip to the value of slip at which the mu-slip curve reaches a peak value, this being the optimum value for minimum braking distance (see note below.).

- **Note:** In an actual vehicle, the slip cannot be measured directly, so this control algorithm is not practical. It is used in this example to illustrate the conceptual construction of such a simulation model. The real engineering value of a simulation like this is to show the potential of the control concept prior to addressing the specific issues of implementation.

**Creating a Temporary Directory for the Example**

During this example, Simulink generates files in the current working directory. If you do not want to generate files in this directory, change the working directory to a suitable directory:

```matlab
origdir = cd(tempdir);
```

**Opening the Model**

To open this model type `sldemo_absbrake` in MATLAB terminal (or click on the hyperlink if you are using MATLAB Help).
**Figure 1: Anti-Lock Braking (ABS) Model**

Double click on the 'Wheel Speed' subsystem in the model window to open it. Given the wheel slip, the desired wheel slip, and the tire torque, this subsystem calculates the wheel angular speed.
To control the rate of change of brake pressure, the model subtracts actual slip from the desired slip and feeds this signal into a bang-bang control (+1 or -1, depending on the sign of the error; see Figure 2). This on/off rate passes through a first-order lag that represents the delay associated with the hydraulic lines of the brake system. The model then integrates the filtered rate to yield the actual brake pressure. The resulting signal, multiplied by the piston area and radius with respect to the wheel \((K_f)\), is the brake torque applied to the wheel.

The model multiplies the frictional force on the wheel by the wheel radius \((R_r)\) to give the accelerating torque of the road surface on the wheel. The brake torque is subtracted to give the net torque on the wheel. Dividing the net torque by the wheel rotational inertia, \(I\), yields the wheel acceleration, which is then integrated to provide wheel velocity. In order to keep the wheel speed and vehicle speed positive, limited integrators are used in this model.
Running the Simulation in ABS Mode

Press the "Play" button on the model toolbar to run the simulation. You can also run the simulation by executing the `sim('sldemo_absbrake')` command in MATLAB. ABS is turned on during this simulation.
Figure 3: Baseline Simulation Results

- Note: The model logs relevant data to MATLAB workspace in a structure called sldemo_absbrake_output. Logged signals have a blue indicator. In this case yout and slp are logged (see the model). Read more about Signal Logging in Simulink Help.

Figure 3 visualizes the ABS simulation results (for default parameters). The first plot in Figure 3 shows the wheel angular velocity and corresponding vehicle angular velocity. This plot shows that the wheel speed stays below vehicle speed without locking up, with vehicle speed going to zero in less than 15 seconds.

Running the Simulation Without ABS

For more meaningful results, consider the vehicle behavior without ABS. At the MATLAB command line, set the model variable ctrl = 0. This disconnects the slip feedback from the controller (see Figure 1), resulting in maximum braking. The results are shown in Figure 4.

ctrl = 0;

Now run the simulation again. This will model braking without ABS.
Figure 4: Maximum braking simulation results (braking without ABS)

Braking With ABS Versus Braking Without ABS

In the upper plot of Figure 4, observe that the wheel locks up in about seven seconds. The braking, from that point on, is applied in a less-than-optimal part of the slip curve. That is, when $\text{slip} = 1$, as seen in the lower plot of Figure 4, the tire is skidding so much on the pavement that the friction force has dropped off.

This is, perhaps, more meaningful in terms of the comparison shown in Figure 5. The distance traveled by the vehicle is plotted for the two cases. Without ABS, the vehicle skids about an extra 100 feet, taking about three seconds longer to come to a stop.
Figure 5: Stopping distance for hard braking with and without ABS

Closing the Model

Close the model. Close the 'Wheel Speed' subsystem. Clear logged data. Change back to the original directory.

cd(origdir);

Conclusions

This model shows how you can use Simulink to simulate a braking system under the action of an ABS controller. The controller in this example is idealized, but you can use any proposed control algorithm in its place to evaluate the system's performance. You can also use the Simulink® Coder™ with Simulink as a valuable tool for rapid prototyping of the proposed algorithm. C code is generated and compiled for the controller hardware to test the concept in a vehicle. This significantly reduces the time needed to prove new ideas by enabling actual testing early in the development cycle.

For a hardware-in-the-loop braking system simulation, you can remove the 'bang-bang' controller and run the equations of motion on real-time hardware to emulate the wheel and vehicle dynamics. You can do this by generating real-time C code for this model using the Simulink Coder. You can then test an actual ABS controller by interfacing it to the real-time hardware, which runs the generated code. In this scenario, the real-time model
would send the wheel speed to the controller, and the controller would send brake action to the model.
Automotive Suspension

This example shows how to model a simplified half-car model that includes an independent front and rear vertical suspension. The model also includes body pitch and bounce degrees of freedom. The example provides a description of the model to show how simulation can be used to investigate ride characteristics. You can use this model in conjunction with a powertrain simulation to investigate longitudinal shuffle resulting from changes in throttle setting.

Analysis and Physics

![Figure 1: A free-body diagram of the half-car model](image)

Figure 1 illustrates the modeled characteristics of the half-car. The front and rear suspension are modeled as spring/damper systems. A more detailed model would include a tire model, and damper nonlinearities such as velocity-dependent damping (with greater damping during rebound than compression). The vehicle body has pitch and bounce degrees of freedom. They are represented in the model by four states: vertical displacement, vertical velocity, pitch angular displacement, and pitch angular velocity. A full model with six degrees of freedom can be implemented using vector algebra blocks to perform axis transformations and force/displacement/velocity calculations. The front suspension influences the bounce (i.e. vertical degree of freedom) according to Equation 1.

Equation 1
\[ F_{\text{front}} = 2K_f(L_f\theta - z) + 2C_f(L_f\dot{\theta} - \dot{z}) \]

Equation 2 contains expressions for the rear suspension.

\[ F_{\text{rear}} = -2K_r(L_r\theta + z) - 2C_r(L_r\dot{\theta} + \dot{z}) \]

Equation 3 contains expressions for the rear suspension.

\[ M_{\text{front}} = -L_f F_{\text{front}} \] (pitch moment due to front suspension)

\[ M_{\text{rear}} = L_r F_{\text{rear}} \] pitch moment due to rear suspension

The forces and moments result in body motion according to Newton's Second Law (see Equation 4).

\[ m_b \ddot{z} = F_{\text{front}} + F_{\text{rear}} - m_bg \]

\[ I_{yy} \ddot{\theta} = M_{\text{front}} + M_{\text{rear}} + M_y \]

\[ m_b = \text{body mass} \]

\[ M_y = \text{pitch moment induced by vehicle acceleration} \]
\[ I_{yy} = \text{body moment of inertia about gravity center} \]

**The Model**

To open this model, type `sldemo_suspn` in MATLAB® Command Window. Figure 2 shows the top-level diagram of the suspension model.

**Vehicle Suspension Model**

![Vehicle Suspension Model Diagram]

**Figure 2**: Top-level diagram of the suspension model

The suspension model shown in Figure 2 has two inputs, and both input blocks are blue on the model diagram. The first input is the road height. A step input here corresponds to the vehicle driving over a road surface with a step change in height. The second input is a horizontal force acting through the center of the wheels that results from braking or acceleration maneuvers. This input appears only as a moment about the pitch axis because the longitudinal body motion is not modeled.
The spring/damper subsystem that models the front and rear suspensions is shown in Figure 3. Right click on the Front/Rear Suspension block and select Mask > Look Under Mask to see the front/rear suspension subsystem. The suspension subsystems are used to model Equations 1-3. The equations are implemented directly in the Simulink® diagram through the straightforward use of Gain and Summation blocks.

The differences between front and rear are accounted for as follows. Because the subsystem is a masked block, a different data set (L, K and C) can be entered for each instance. Furthermore, L is thought of as the Cartesian coordinate x, being negative or positive with respect to the origin, or center of gravity. Thus, $K_f$, $C_f$, and $-L_f$ are used for the front suspension block whereas $K_r$, $C_r$, and $L_r$ are used for the rear suspension block.

**Running the Simulation**

To run this model, press the Play button on the toolbar in the model window. Initial conditions are loaded into the model workspace from the `sldemo_suspdat.m` file (open this file to view its contents). To see the contents of the model workspace, in the Simulink Editor, on the **Modeling** tab, under **Design**, select **Model Explorer**. In the Model Explorer, look under the contents of the `sldemo_suspn` model and select "Model..."
Workspace. Loading initial conditions in the model workspace prevents any accidental modifications of parameters and keeps MATLAB workspace clean.

Note that the model logs relevant data to MATLAB workspace in a data structure called sldemo_suspn_output. Type the name of the structure to see what data it contains. Read more about signal logging in Simulink Help.
Simulation Model Simulation Results

- $\dot{\theta}$ (rad/\(\text{sec}\))
- $\dot{z}$ (m/\(\text{sec}\))

- $F_{\text{react}}$ (N)
  - reaction force at front wheels

- $h$ (m)
  - road height

- $M_g$ (units)
  - moment due to vehicle accel/decel

Time (sec)
Figure 4: Simulation results

Simulation results are displayed in Figure 4. The results are plotted by the sldemo_suspgraph.m (you can open this file to see how it is done). The default initial conditions are given in Table 1 below.

Table 1: Default initial conditions

Lf = 0.9;    % front hub displacement from body gravity center (m)
Lr = 1.2;    % rear hub displacement from body gravity center (m)
Mb = 1200;   % body mass (kg)
Iyy = 2100;  % body moment of inertia about y-axis in (kg m^2)
kf = 28000;  % front suspension stiffness in (N/m)
kr = 21000;  % rear suspension stiffness in (N/m)
cia = 2500; % front suspension damping in (N sec/m)
cr = 2000;   % rear suspension damping in (N sec/m)

Closing the Model

Close the model and delete generated data from MATLAB workspace.

Conclusions

This model allows you to simulate the effects of changing the suspension damping and stiffness, thereby investigating the tradeoff between comfort and performance. In general, racing cars have very stiff springs with a high damping factor, whereas passenger vehicles have softer springs and a more oscillatory response.
**Modeling an Automatic Transmission Controller**

This example shows how to model an automotive drivetrain with Simulink®. Stateflow® enhances the Simulink model with its representation of the transmission control logic. Simulink provides a powerful environment for the modeling and simulation of dynamic systems and processes. In many systems, though, supervisory functions like changing modes or invoking new gain schedules must respond to events that may occur and conditions that develop over time. As a result, the environment requires a language capable of managing these multiple modes and developing conditions. In the following example, Stateflow shows its strength in this capacity by performing the function of gear selection in an automatic transmission. This function is combined with the drivetrain dynamics in a natural and intuitive manner by incorporating a Stateflow block in the Simulink block diagram.

**Analysis and Physics**

Figure 1 shows the power flow in a typical automotive drivetrain. Nonlinear ordinary differential equations model the engine, four-speed automatic transmission, and vehicle. The model discussed in this example directly implements the blocks from Figure 1 as modular Simulink subsystems. On the other hand, the logic and decisions made in the Transmission Control Unit (TCU) do not lend themselves to well-formulated equations. TCU is better suited for a Stateflow representation. Stateflow monitors the events which correspond to important relationships within the system and takes the appropriate action as they occur.

![Figure 1: Generic block diagram for a drivetrain system](image)
The throttle opening is one of the inputs to the engine. The engine is connected to the impeller of the torque converter which couples it to the transmission (see Equation 1).

**Equation 1**

\[ I_{ei} \dot{N}_e = T_e - T_i \]

- \( N_e \) = engine speed (RPM)
- \( I_{ei} \) = moment of inertia of the engine and the impeller
- \( T_e, T_i \) = engine and impeller torque

The input-output characteristics of the torque converter can be expressed as functions of the engine speed and the turbine speed. In this example, the direction of power flow is always assumed to be from the impeller to the turbine (see Equation 2).

**Equation 2**

\[ T_i = \frac{N_e^2}{K^2} \]

- \( K = f_2 \frac{N_{in}}{N_e} \) = K-factor (capacity)
- \( N_{in} \) = speed of turbine (torque converter output) = transmission input speed (RPM)
- \( R_{TQ} = f_3 \frac{N_{in}}{N_e} \) = torque ratio

The transmission model is implemented via static gear ratios, assuming small shift times (see Equation 3).

**Equation 3**

\[ R_{TR} = f_4 \text{(gear)} \] = transmission ratio

\[ T_{out} = R_{TR} T_{in} \]

\[ N_{in} = R_{TR} N_{out} \]
The final drive, inertia, and a dynamically varying load constitute the vehicle dynamics (see Equation 4).

**Equation 4**

\[
I_v \dot{N}_w = R_{fd} (T_{out} - T_{load})
\]

\(I_v\) = vehicle inertia
\(N_w\) = wheel speed (RPM)
\(R_{fd}\) = final drive ratio
\(T_{load}\) = load torque

The load torque includes both the road load and brake torque. The road load is the sum of frictional and aerodynamic losses (see Equation 5).

**Equation 5**

\[
T_{load} = sgn(mph)(R_{load0} + R_{load2} mph^2 + T_{brake})
\]

\(R_{load0}, R_{load2}\) = friction and aerodynamic drag coefficients
\(T_{load}, T_{brake}\) = load and brake torques
\(mph\) = vehicle linear velocity

The model programs the shift points for the transmission according to the schedule shown in Figure 2. For a given throttle in a given gear, there is a unique vehicle speed at which an upshift takes place. The simulation operates similarly for a downshift.
**Figure 2:** Shift schedule

**Modeling**

To open this model type `sldemo_autotrans` in MATLAB® terminal. Initial conditions are set in the Model Workspace.

The top-level diagram of the model is shown in Figure 3. To run the simulation, press the Play button on the toolbar in the model window. Note that the model logs relevant data to
MATLAB Workspace in a data structure called sldemo_autotrans_output. Logged signals have a blue indicator (see Figure 3). After you run the simulation, you can view the components of the data structure by typing sldemo_autotrans_output in MATLAB Command Window. Also note that the units appear on the subsystem icons and signal lines. To learn more about units in Simulink see Simulink Units.

Figure 3: Model diagram and sample simulation results (passing maneuver)

Modeling

The Simulink model shown in Figure 3 is composed of modules which represent the engine, transmission, and the vehicle, with an additional shift logic block to control the transmission ratio. User inputs to the model are in the form of throttle (given in percent) and brake torque (given in ft-lb). The user inputs throttle and brake torques using the ManeuversGUI interface.

The Engine subsystem consists of a two-dimensional table that interpolates engine torque versus throttle and engine speed. Figure 4 shows the composite Engine subsystem. Double click on this subsystem in the model to view its structure.
Figure 4: Engine subsystem

The TorqueConverter and the TransmissionRatio blocks make up the Transmission subsystem, as shown in Figure 5. Double click on the Transmission subsystem in the model window to view its components.

Figure 5: Transmission subsystem

The TorqueConverter is a masked subsystem, which implements Equation 2. To open this subsystem, right click on it and select Mask > Look Under Mask from the drop-down menu. The mask requires a vector of speed ratios ($\frac{N_{in}}{N_e}$) and vectors of K-factor ($f_2$) and torque ratio ($f_3$). Figure 6 shows the implementation of the TorqueConverter subsystem.
Figure 6: Torque converter subsystem

The transmission ratio block determines the ratio shown in Table 1 and computes the transmission output torque and input speed, as indicated in Equation 3. Figure 7 shows the block diagram for the subsystem that realizes this ratio in torque and speed.

Table 1: Transmission gear ratios

<table>
<thead>
<tr>
<th>gear</th>
<th>$R_{tr} = \frac{N_{in}}{N_e}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2.393</td>
</tr>
<tr>
<td>2</td>
<td>1.450</td>
</tr>
<tr>
<td>3</td>
<td>1.000</td>
</tr>
<tr>
<td>4</td>
<td>0.677</td>
</tr>
</tbody>
</table>
Figure 7: Transmission gear ratio subsystem

The Stateflow block labeled ShiftLogic implements gear selection for the transmission. Double click on ShiftLogic in the model window to open the Stateflow diagram. The Model Explorer is utilized to define the inputs as throttle and vehicle speed and the output as the desired gear number. Two dashed AND states keep track of the gear state and the state of the gear selection process. The overall chart is executed as a discrete-time system, sampled every 40 milliseconds. The Stateflow diagram shown in Figure 8 illustrates the functionality of the block.
Figure 8: Stateflow diagram of the transmission shift logic

The shift logic behavior can be observed during simulation by enabling animation in the Stateflow debugger. The selection_state (always active) begins by performing the computations indicated in its during function. The model computes the upshift and downshift speed thresholds as a function of the instantaneous values of gear and throttle. While in steady_state, the model compares these values to the present vehicle speed to determine if a shift is required. If so, it enters one of the confirm states (upshifting or downshifting), which records the time of entry.

If the vehicle speed no longer satisfies the shift condition, while in the confirm state, the model ignores the shift and it transitions back to steady_state. This prevents
extraneous shifts due to noise conditions. If the shift condition remains valid for a duration of TWAIT ticks, the model transitions through the lower junction and, depending on the current gear, it broadcasts one of the shift events. Subsequently, the model again activates steady_state after a transition through one of the central junctions. The shift event, which is broadcast to the gear_selection state, activates a transition to the appropriate new gear.

For example, if the vehicle is moving along in second gear with 25% throttle, the state second is active within gear_state, and steady_state is active in the selection_state. The during function of the latter, finds that an upshift should take place when the vehicle exceeds 30 mph. At the moment this becomes true, the model enters the upshifting state. While in this state, if the vehicle speed remains above 30 mph for TWAIT ticks, the model satisfies the transition condition leading down to the lower right junction. This also satisfies the condition \(|\text{gear} == 2|\) on the transition leading from here to steady_state, so the model now takes the overall transition from upshifting to steady_state and broadcasts the event UP as a transition action. Consequently, the transition from second to third is taken in gear_state which completes the shift logic.

The Vehicle subsystem (Figure 9) uses the net torque to compute the acceleration and integrate it to compute the vehicle speed, per Equation 4 and Equation 5. The Vehicle subsystem is masked. To see the structure of the Vehicle block, right click on it and select Mask > Look Under Mask from the drop-down menu. The parameters entered in the mask menu are the final drive ratio, the polynomial coefficients for drag friction and aerodynamic drag, the wheel radius, vehicle inertia, and initial transmission output speed.
**Figure 9:** Vehicle subsystem (masked)

**Results**

The engine torque map, and torque converter characteristics used in the simulations are shown in Figure 10 and Figure 11.

**Figure 10:** Engine torque map
The first simulation (passing maneuver) uses the throttle schedule given in Table 2 (this data is interpolated linearly).

**Table 2:** Throttle schedule for first simulation (passing maneuver)

<table>
<thead>
<tr>
<th>Time (sec)</th>
<th>Throttle (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>60</td>
</tr>
<tr>
<td>14.9</td>
<td>40</td>
</tr>
<tr>
<td>15</td>
<td>100</td>
</tr>
<tr>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>200</td>
<td>0</td>
</tr>
</tbody>
</table>

The first column corresponds to time; the second column corresponds to throttle opening in percent. In this case no brake is applied (brake torque is zero). The vehicle speed starts...
at zero and the engine at 1000 RPM. Figure 12 shows the plot for the baseline results, using the default parameters. As the driver steps to 60% throttle at $t=0$, the engine immediately responds by more than doubling its speed. This brings about a low speed ratio across the torque converter and, hence, a large torque ratio (see Figure 6 and Figure 11. The vehicle accelerates quickly (no tire slip is modeled) and both the engine and the vehicle gain speed until about $t = 2$ sec, at which time a 1-2 upshift occurs. The engine speed characteristically drops abruptly, then resumes its acceleration. The 2-3 and 3-4 upshifts take place at about four and eight seconds, respectively. Notice that the vehicle speed remains much smoother due to its large inertia.

**Figure 12:** Passing maneuver simulation time history

At $t=15$ sec, the driver steps the throttle to 100% as might be typical of a passing maneuver. The transmission downshifts to third gear and the engine jumps from about
2600 RPM to about 3700 RPM. The engine torque thus increases somewhat, as well as the mechanical advantage of the transmission. With continued heavy throttle, the vehicle accelerates to about 100 mph and then shifts into overdrive at about \( t = 21 \text{ sec} \). The vehicle cruises along in fourth gear for the remainder of the simulation. Double click on the ManeuversGUI block and use the graphical interface to vary the throttle and brake history.

**Closing the Model**

Close the model, clear generated data.

**Conclusions**

One can easily enhance this basic system in a modular manner, for example, by replacing the engine or transmission with a more complex model. We can thus build up large systems within this structure via step-wise refinement. The seamless integration of Stateflow control logic with Simulink signal processing enables the construction of a model which is both efficient and visually intuitive.
Vehicle Electrical System

This example shows how to simulate the electrical system of a vehicle using Simulink® and Simscape™ Power Systems™.

System Components

The system simulated consists of a few components. The main power source is the internal combustion (IC) engine which drives the shaft of the alternator. The alternator produces the AC power. The diode trio rectifies the AC current generated from the alternator to DC current which is applied to the alternator field. The indicator lamp lights when the system is turned on and the alternator is not producing power. The rectifier bridge converts the AC power into DC power. The smoothing capacitor, voltage sensor, battery, and vehicle loads are connected to the DC bus.

The alternator is a 3-phase synchronous machine with its field current regulated to give control over the DC bus voltage. This is simulated by using the Synchronous Machine Round Rotor model.

Figure 1: The vehicle electrical system.
The 3-phase AC output of the alternator is fed into the 6-pulse rectifier bridge to give the DC voltage required to charge the car battery and to supply the balance of the electrical system of the car.

**Operation of the System**

When the IC engine is off, the battery supplies the current to the alternator field winding. The indicator lamp is turned on.

When the IC engine runs, the alternator will start to produce voltage and the diode trio will conduct and let the current flow to the alternator field winding. The indicator lamp is then turned off.

When the DC bus voltage reaches the upper voltage limit (i.e., 14.5 V in the model), the current flowing to the alternator field is bypassed. As a result, the DC bus voltage decreases. If it reaches the lower voltage limit (i.e., 13.5 V in the model), the diode trio is reconnected to the field winding.
Simulating Automatic Climate Control Systems

This example shows how to simulate the working of an automatic climate control system in a car using Simulink® and Stateflow®. You can enter a temperature value you would like the air in the car to reach by double clicking the User Setpoint in Celsius Block and entering the temperature value. You can also set the External Temperature in Celsius in a similar way. The numerical display on the right-hand side of the model shows the reading of a temperature sensor placed behind the driver's head. This is the temperature that the driver should be feeling. When the model is run and the climate control is active, it is this display box whose value changes to show the change of temperature in the car.

Figure 1: The automatic climate control system.

The Stateflow® Controller

The supervisory controller is implemented in Stateflow. Double clicking the Stateflow chart shows how this supervisory control logic has been formulated.

The Heater_AC state shows that when you enter a setpoint temperature that is greater than the current temperature in the car by at least 0.5 deg C, the heater system will be
switched on. The heater will remain active until the current temperature in the car is within 0.5 deg of the setpoint temperature. Similarly, when the user enters a setpoint that is 0.5 deg C (or more) lower than the current car temperature, the air conditioner is turned on and stays active until the air temperature in the car is within 0.5 deg C of the setpoint temperature, after which, the system will switch off. A dead band of 0.5 deg has been implemented to avoid the problem of continuous switching.

In the **Blower** State, the larger the difference between the setpoint temperature and the current temperature, the harder the fan blows. This ensures that the temperature will reach the required value in a reasonable amount of time, despite the temperature difference. Once again, when the air temperature in the car is within 0.5 deg C of the setpoint temperature, the system will switch off.

The Air Distribution(**AirDist**) and Recycling Air States(**Recyc_Air**) are controlled by the two switches that trigger the Stateflow chart. An internal transition has been implemented within these two states to facilitate effective defrosting of the windows when required. When the defrost state is activated, the recycling air is turned off.
Figure 2: The supervisory control logic in Stateflow.

**Heater and Air Conditioner Models**

The heater model was built from the equation for a heater exchanger shown below:

\[ T_{out} = T_s - (T_s - T_{in})e^{\left(-\frac{\pi D L hc}{m \_dot \_Cp}\right)} \]

Where:

- \( T_s = \) constant (radiator wall temperature)
- \( D = 0.004m \) (channel diameter)
• L = 0.05m (radiator thickness)
• N = 30000 (Number of channels)
• k = 0.026 W/mK = constant (thermal conductivity of air)
• Cp = 1007 J/kgK = constant (specific heat of air)
• Laminar flow (hc = 3.66(k/D) = 23.8 W/m2K )

In addition, the effect of the heater flap is taken into account. Similar to the operation of
the blower, the greater the temperature difference between the required setpoint
temperature and the current temperature in the car, the more the heater flap is opened
and the greater the heating effect.

The air conditioner model was built from the equation below:

\[ y \cdot (w \cdot T_{comp}) = m_{dot} \cdot (h_4 - h_1) \]

Where:

• y = efficiency
• m_dot = mass flow rate
• w = speed of the engine
• T_{comp} = compressor torque
• h4, h1 = enthalpy

Here we have bang-bang control of the A/C system where the temperature of the air that
exits the A/C is determined by the engine speed and compressor torque.
Figure 3: Heater control subsystem.

Heat Exchanger Equation:
T_{out} = T_s - (T_s - T_{in})e^{[(-\pi D L h_c)/m_{dot} C_p)]}

Assumptions:
Ts = constant (radiator wall temperature)
D = 0.004m (channel diameter)
L = 0.05m (radiator thickness)
N = 30000 (No of channels)
k = 0.026 W/mK = constant (thermal conductivity of air)
C_p = 1007 J/kgK = constant (specific heat of air)
Laminar flow (hc = 3.66(k/D) = 23.8 W/m2K)
Heat Transfer in the Cabin

The temperature of the air felt by the driver is affected by all of these factors:

- The temperature of the air exiting the vents
- The temperature of the outside air
- The number of people in the car

These factors are inputs into the thermodynamic model of the interior of the cabin. We take into account the temperature of the air exiting the vents by calculating the difference between the vent air temperature and the current temperature inside the car and multiplying it by the fan speed proportion (mass flow rate). Then 100W of energy is added per person in the car. Lastly, the difference between the temperature of the outside air and the interior air temperature is multiplied by a lesser mass flow rate to account for the air radiating into the car from the outside.

Figure 4: A/C control subsystem.
The output of the interior dynamics model is fed to the display block as a measure of the temperature read by a sensor placed behind the driver's head.
Vehicle Electrical and Climate Control Systems

This example shows how to interface the vehicle climate control system with a model of the electrical system to examine the loading effects of the climate control system on the entire electrical system of the car.

**Figure 1:** Vehicle Electrical and Climate Control System

**The Climate Control System**

Double clicking on the ClimateControlSystem subsystem will open the model of the climate control system. Here the user can enter the temperature value they would like the air in the car to reach by double clicking on the USER SETPOINT IN CELSIUS Block and entering the value into the dialog box. The EXTERNAL TEMPERATURE IN CELSIUS can also be set by the user in a similar way. The numerical display on the right hand side of the model shows the reading of a temperature sensor placed behind the driver's head. This is the temperature that the driver should be feeling. When the model is run and the climate control is active, it is this display box whose value changes showing the change of temperature in the car.
Figure 2: The automatic climate control system.

The Stateflow® Controller

The control of the system is implemented in Stateflow®. Double clicking on the Stateflow chart will show how this supervisory control logic has been formulated.

The **Heater_AC** state shows that when the user enters a setpoint temperature which greater than the current temperature in the car by at least 0.5 deg C, the heater system will be switched on. The heater will remain active until the current temperature in the car reaches to within 0.5 deg of the setpoint temperature. Similarly, when the user enters a setpoint which is 0.5 deg C (or more) lower than the current car temperature, the Air Conditioner is turned on and stays active until the temperature of the air in the car reaches to within 0.5 deg C of the setpoint temperature. After which, the system will switch off. The dead band of 0.5 deg has been implemented to avoid the problem of continuous switching.

In the **Blower** State, the larger the difference between the setpoint temperature and the current temperature, the harder the fan blows. This ensures that the temperature will reach the required value in a reasonable amount of time, despite the temperature
difference. Once again, when the temperature of the air in the car reaches to within 0.5 deg C of the setpoint temperature, the system will switch off.

The Air Distribution(AirDist) and Recycling Air States(Recyc_Air) are controlled by the two switches that trigger the Stateflow chart. An internal transition has been implemented within these two states to facilitate effective defrosting of the windows when required. When the defrost state is activated, the recycling air is turned off.

**Figure 3:** The supervisory control logic in Stateflow.

**Heater and Air Conditioner Models**

The heater model was built from the equation for a heat exchanger shown below:

\[ T_{out} = T_s - (T_s - T_{in})e^{\frac{-\pi D L h c}{m_{dot} C_p}}} \]
Where:
- \( Ts = \text{constant (radiator wall temperature)} \)
- \( D = 0.004\text{m (channel diameter)} \)
- \( L = 0.05\text{m (radiator thickness)} \)
- \( N = 30000 \text{ (Number of channels)} \)
- \( k = 0.026 \text{ W/mK = constant (thermal conductivity of air)} \)
- \( \text{Cp} = 1007 \text{ J/kgK = constant (specific heat of air)} \)
- \( \text{Laminar flow (hc} = 3.66(k/D) = 23.8 \text{ W/m2K} \)

In addition, the effect of the heater flap is taken into account. Similar to the operation of the blower, the greater the temperature difference between the required setpoint temperature and the current temperature in the car, the more the heater flap is opened and the greater the heating effect.

The Air Conditioner system is one of the two places where the climate control model interfaces with the car's electrical system model. The compressor loads the engine of the car when the A/C system is active. The final temperature to exit from the A/C is calculated as follows:

\[
y^* (w^* T_{comp}) = m_{dot}^* (h_4 - h_1)\]

Where:
- \( y = \text{efficiency} \)
- \( m_{dot} = \text{mass flow rate} \)
- \( w = \text{speed of the engine} \)
- \( T_{comp} = \text{compressor torque} \)
- \( h_4, h_1 = \text{enthalpy} \)

Here we have bang-bang control of the A/C system where the temperature of the air that exits the A/C is determined by the engine speed and compressor torque.
**Figure 4:** Heater control subsystem.
Figure 5: A/C control subsystem.

Heat Transfer in the Cabin

The temperature of the air felt by the driver is affected by all of these factors:

- The temperature of the air exiting the vents
- The temperature of the outside air
- The number of people in the car

These factors are inputs into the thermodynamic model of the interior of the cabin. We take into account the temperature of the air exiting the vents by calculating the difference between the vent air temperature and the current temperature inside the car and multiplying it by the fan speed proportion (mass flow rate). Then 100W of energy is added per person in the car. Lastly, the difference between the temperature of the outside
air and the interior air temperature is multiplied by a lesser mass flow rate to account for the air radiating into the car from the outside.

The output of the interior dynamics model is fed to the display block as a measure of the temperature read by a sensor placed behind the driver's head.

**The Electrical System**

This electrical system models the car at idle speed. The PID controllers ensure that the car's alternator (modeled by a synchronous machine which has its field current regulated to control the output voltage) is also operating at the required speed. The alternator output is then fed through a 3-phase 6-pulse rectifier bridge to supply the voltage needed to charge the battery which supplies the voltage for the car's DC bus.

The fan used in the climate control system is fed off this DC bus as are the windscreen wipers, radio etc. As the difference between the setpoint temperature and the current temperature in the car drops, so does the fan speed and therefore so does the loading on the DC bus. The inclusion of feedback in the electrical system regulates the DC bus voltage.

The additional model of the car's electrical system allows for the changing of the engine speed. Changing the engine speed shows the effect on the DC bus voltage.
Figure 6: The electrical system
Power Window Control Project

This example shows how you can use MathWorks® software and the Model-Based Development process to go from concept to implementation for a power window system for an automobile.

Overview of Design Requirements

In this example, consider the passenger-side power window system of an automobile. Note a critical aspect of this system is that it can never exert a force of more than 100 N on an object when closing the window, for example:

![Diagram of power window system](image)

When the model detects such an object, it must lower the window by about 10 centimeters.

For more information on design requirements see power window controller documentation

Using Model-Based Development and Large Scale Modeling

This example uses Model-Based Design and large scale modeling techniques such as,

- **Model Blocks** to separate the hierarchy into separate models.
- **Variant Subsystems Blocks** to model and switch between different design choices.
- **Libraries** to capture algorithms for reuse in variant subsystems.
- **Simulink® Project** to manage the files required for the system development.
Opening Power Window Control Project

Run the following command to create and open a working copy of the project files for this example.

`slexPowerWindowStart`

Exploring the Project

Upon visual inspection of the project, you can see features from Simulink Project used to organize the example. These features are

- Folders
- File Classification
- Shortcuts

Folders
The project is organized into the following folders:

- **configureModel** - contains the MATLAB® files controlling the main system model variant configuration.
- **data** - contains the images required by the project.
- **hmi** - contains the files to animate the response of the power window.
- **model** - contains the main system model, controller model, models for testing the controller, and libraries which support these models.
- **task** - contains the MATLAB files that simulate the model for the different model configurations and generate coverage reports for the controller.
- **utilities** - contains the MATLAB files to initialize the model, to generate spreadsheet input, to add data to the generated spreadsheet and to manage the project's environment at startup and shutdown.

**File Classification**

Files in Simulink Projects have different classifications visible in Labels Pane. Each label describes the specific role a file contributes to the body of the project. In this project, new classifications were added, including:

- **Configuration** - the files that configure the project or model.
- **PrjConfig** - the files that configure the project by adding its files to the path at startup and removing them at shutdown.
- **DesignConfig** - the files that determine which model configuration is active at a given time.
- **Design** - the main system model and its referenced control model.
- **DesignSupport** - files such as libraries, data, and model simulation.
- **Simulation** - the files that simulate the model for a specific configuration.
- **Test** - the control coverage, the control interaction, and the test harness models.
- **Visualization** - the files that animate the motion of the power window.

**Shortcuts**

Project shortcuts are used to obtain quick access to project files that are used most often. Some shortcuts contain general tasks such as adding the project to the path at startup and removing it at shutdown. In addition, project shortcut groups help organize the shortcuts. The new shortcut groups are
Interactive Testing - contains files used for interactive testing of the controller.
Main Model - contains file for top level Simulink model.
Model Coverage - contains files used for model coverage of the controller.
Simulation - contains files used for simulation of model variant configurations.

Exploring Simulink Models in Project

The Simulink Models for this project are located in the model folder. The models of interest are

- Main System Model
- Models for Testing

Main System Model

In this example under the model folder, the main system model is slexPowerWindowExample. This model consists of the driver and passenger switch blocks, which generate the inputs to the system. The inputs then go through the control system model, which validates the states of the passenger and driver inputs. The control system block also determines if there is an obstacle blocking the window's path. The referenced controller produces the window motion command signals sent to the active variant of the window system. The window system output is the feedback to the control system block.

To visualize the results of simulation, Simulation Data Inspector (SDI) logs the output data and Simulink 3D Animation™ animates the window's motion.

Model Variants

The main system model in this project uses the Variant Subsystem block to allow for multiple implementations of within a subsystem. The active implementation can be programmatically changed prior to simulation. In the main model, there are four Variant Subsystem blocks each with variant choices that can be programmatically modified. Those four variant subsystems are:

- slexPowerWindowExample/driver_switch
- slexPowerWindowExample/passenger_switch
- slexPowerWindowExample/window_system
- slexPowerWindowExample/power_window_control_system/detect_obstacle_endstop
Each variant choice is associated with a variant control. The variant choice is active when its variant control evaluates to true.

Since there are four programmatically modifiable Variant Subsystem blocks in the main model, there are MATLAB files to control the combinations of variant choices to create model variant configurations which are found under the DesignConfig classification. Available model variant configurations are

- Power Window Controller Hybrid System Model
- Power Window Controller and Detailed Plant Model
- Power Window Controller With Data Acquisition Effects
- Power Window Controller With Controller Area Network (CAN) Communication

**Power Window Controller Hybrid System Model**

This model variant uses Stateflow® and Simulink to model both discrete event reactive behavior and continuous time behavior. It uses a low order plant model to validate the roll-up and roll-down behavior. You can simulate this variant configuration using the SimHybridPlantLowOrder shortcut. This shortcut activates only the variant subsystem corresponding to this model configuration. Since this model does not take into account power effects, the only output logged is position. Simulation Data Inspector (SDI) displays the logged position data.

**Power Window Controller and Detailed Plant Model**

This model variant shows a more detailed plant model that includes power effects in the electrical and mechanical domains is used to validate that the force exerted by the window on a trapped object never exceeds 100 N. This model variant requires Simscape™ Multibody™ and Simscape Power Systems™ products to be installed. You can simulate this variant configuration using the SimHybridPlantPowerEffects shortcut. Unlike the previous variant model, this variant configuration takes into account the power effects. SDI displays the logged data from armature current, position, and force exerted by the power window.

**Power Window Controller With Data Acquisition Effects**

This model variant shows additional effects due to implementation that affect the control. The phenomena included are signal conditioning to measure the armature current and quantization of the measurement. This model variant requires Simscape Multibody, Simscape Power Systems, DSP System Toolbox™, and Fixed-Point Designer™ products to be installed. You can simulate this variant configuration using the
SimHybridPlantPowerEffects+ControlDAQEffects shortcut. Like the previous model, SDI displays the logged data from armature current, position, and force exerted by the power window.

**Power Window Controller With CAN Communication**

This model variant shows the use of a CAN to communicate the commands to control window movement. The switches that may be located in the center console of the vehicle and that produce the commands are included in this model variant. This model variant requires Simscape Multibody, Simscape Power Systems, DSP System Toolbox, and Fixed-Point Designer products to be installed. You can simulate this variant configuration on a machine running windows OS using the|SimCANCommunication| shortcut.

**Models for Testing**

To test the state machine that controls the power window, you can run the project shortcuts for testing. Available model shortcuts for testing the controller are

- InteractiveExample
- CoverageExample
- IncreaseCoverageExample

**InteractiveExample**

This model shortcut opens the model, slexPowerWindowCntlInteract. This model contains the power window controller which is a state machine. This model also contains inputs to the controller which are selected with manual switch blocks.

The power window controller has four external inputs:

- Passenger Input
- Driver Input
- Window Frame Endstops
- Obstacle Present

**Passenger Input**

This input consists of a vector with three elements:

- `neutral`: the passenger control switch is not depressed
• **up**: the passenger control switch generates the up signal
• **down**: the passenger control switch generates the down signal

*Driver Input*

This input consists of a vector with three elements:

• **neutral**: the driver control switch is not depressed
• **up**: the driver control switch generates the up signal
• **down**: the driver control switch generates the down signal

*Window Frame Endstops*

This input consists of a vector with two elements:

• **0**: window moves freely between top or bottom
• **1**: window is stuck at the top or bottom because of physical limitations

*Obstacle Present*

This input consists of a vector with two elements:

• **0**: window moves freely between top or bottom
• **1**: window has obstacle within its frame

You can interactively test the controller by simulating the model and selecting the desired combination of inputs via the manual switch blocks. After making the selection of inputs, you can verify the internal controller state and controller output against the desired result for this specific set of inputs.

*CoverageExample*

This model shortcut opens the model, slexPowerWindowCntlCoverage. This model contains the power window controller which is a state machine. This model also contains inputs to the controller which are repeating sequence blocks.

You can use the Simulink Coverage (TM) Model Coverage tool to validate the discrete event control of the window. The Model Coverage tool helps determine the extent to which a model test case exercises the conditional branches of the controller. It also helps evaluate whether all transitions in the discrete event control are taken given the test case we run. It also evaluates whether all clauses in a condition that enables a particular
transition have become true. One transition may be enabled by multiple clauses, e.g., the transition from emergency back to neutral occurs when either 100 ticks have occurred or when the endstop is reached.

**IncreaseCoverageExample**

This model shortcut opens the model, slexPowerWindowCntlCoverageIncrease. This model contains the power window controller that is a state machine. This model also contains a From Spreadsheet block that provides multiple sets of inputs to the controller. These input sets combine with the one from the CoverageExample model to exercise more of the logic in the power window controller.

These input sets are:

- **Logged**: Logged from the CoverageExample.
- **LoggedObstacleOffEndStopOn**: Logged from the CoverageExample with ability to hit endstop.
- **LoggedObstacleOnEndStopOff**: Logged from the CoverageExample with obstacle in window.
- **LoggedObstacleOnEndStopOn**: Logged from the CoverageExample with obstacle in window and ability to hit endstop.
- **DriverLoggedPassengerNeutral**: Logged from the CoverageExample for only the driver. Passenger takes no action.
- **DriverDownPassengerNeutral**: Driver lowering window. Passenger takes no action.
- **DriverUpPassengerNeutral**: Driver raising window. Passenger takes no action.
- **DriverAutoDownPassengerNeutral**: Driver lowering window for 1 second (auto-down). Passenger takes no action.
- **DriverAutoUpPassengerNeutral**: Driver raising window for 1 second (auto-up). Passenger takes no action.
- **PassengerAutoDownDriverNeutral**: Passenger lowering window for 1 second (auto-down). Driver takes no action.
- **PassengerAutoUpDriverNeutral**: Passenger raising window for 1 second (auto-up). Driver takes no action.

The model coverage shortcut, GenerateIncreasedCoverage, uses the multiple input sets with the Simulink Coverage Model Coverage tool to validate the discrete event control of the window and generate a coverage report for the multiple input sets. The Model Coverage tool helps determine the extent to which a model test case exercises the
conditional branches of the controller. It also helps evaluate if all transitions in the
discrete event control have been taken into account given the input sets we run.

**More Information on Power Window Controller**
See power window controller documentation

**More Information on Simulink® Project**
See Simulink Project documentation
Developing the Apollo Lunar Module Digital Autopilot

"Working on the design of the Lunar Module digital autopilot was the highlight of my career as an engineer. When Neil Armstrong stepped off the LM (Lunar Module) onto the moon's surface, every engineer who contributed to the Apollo program felt a sense of pride and accomplishment. We had succeeded in our goal. We had developed technology that never existed before, and through hard work and meticulous attention to detail, we had created a system that worked flawlessly." -Richard J. Gran, *The Apollo 11 Moon Landing: Spacecraft Design Then and Now*

This example shows how Richard and the other engineers who worked on the Apollo Lunar Module digital autopilot design team could have done it using Simulink® and Aerospace Blockset™ if they had been available in 1961.

**Model Description**

Developing the autopilot in Simulink takes a fraction of the time it took for the original design of the Apollo Lunar Module autopilot.
The Reaction Jet Control subsystem models the digital autopilot design proposed (and implemented) by MIT Instrumentation Laboratories (MIT IL), now called Draper Labs. A Stateflow® diagram in the model specifies the logic that implements the phase-plane control algorithm described in the technical article *The Apollo 11 Moon Landing: Spacecraft Design Then and Now*. Depending on which region of the diagram the Lunar Module is executing, the Stateflow diagram is in either a Fire_region or a Coast_region. Note, the transitions between these different regions depend on certain parameters. The Stateflow diagram determines whether to transition to another state and then computes which reaction jets to fire.
Translational and rotational dynamics of the Lunar Module are approximated in the Lunar Module Dynamics subsystem. Access various visualization methods of the Lunar Module states and autopilot performance in the Visualization area of the model, including Simulink scopes, animation with Simulink 3D Animation, and a phase plane plot.

**Interactive Controls**

To interact with the Lunar Module model, vary autopilot settings and Lunar Module initial states in the Commands area. For example, to observe how the digital autopilot design
handles increased initial body rates, use the slider components in the Configure LM Attitude subsystem.

\[ \text{Configure LM Attitude (mask)} \]

- **Pilot Commanded Body Attitude**
  - Specify desired body attitude angles to send to digital autopilot.
  - Pilot commanded attitude (roll, pitch, yaw)
  - Euler angles: \[ \text{euler}_0 + [2.87 \ 5.73 \ 0.606] \]

- **Initial Body Angular Rates**
  - Vary the initial angular rates of the Lunar Module to investigate the response of the digital autopilot design.

\[ \begin{align*}
\text{Roll rate [deg/s]} & : 15.0 \\
\text{Pitch rate [deg/s]} & : 15.0 \\
\text{Yaw rate [deg/s]} & : 15.0 
\end{align*} \]

**Mission Description**

The LM digital autopilot has 3 degrees of freedom. This means that by design, the reaction jet thrusters are configured and commanded to rotate the vehicle without impacting the vehicle's orbital trajectory. Therefore, the translational dynamics in his model are approximated solely via orbit propagation using the Moon Zonal Harmonic Gravity Model from Aerospace Blockset. To demonstrate the digital autopilot design
behavior, the "Descent Orbit Insertion" mission segment, just prior to the initiation of the powered descent, was selected from the Apollo 11 Mission Report.

The Descent Orbit Insertion burn began 101 hours, 36 minutes, and 14 seconds after liftoff and lasted 30 seconds. The burn set the Lunar module on a trajectory to lower its orbit from approximately 60 nautical miles to 50,000 ft over about an hour. At 50,000 ft, the Module initiated its powered descent.

Initialize the model `aero_dap3dof` with the approximate trajectory of the Lunar Module immediately after the descent orbit insertion burn.
### Developing the Apollo Lunar Module Digital Autopilot

<table>
<thead>
<tr>
<th>MissionTime_GMT</th>
<th>MissionPhase</th>
</tr>
</thead>
<tbody>
<tr>
<td>16-Jul-1969 13:32:00</td>
<td>{'Range Zero (lift-off)'}</td>
</tr>
<tr>
<td>20-Jul-1969 19:08:14</td>
<td>{'Descent Orbit Insertion (Engine ignition)'}</td>
</tr>
<tr>
<td>20-Jul-1969 19:08:44</td>
<td>{'Descent Orbit Insertion (Engine cutoff)'}</td>
</tr>
<tr>
<td>20-Jul-1969 20:05:05</td>
<td>{'Powered Descent (Engine ignition)'}</td>
</tr>
</tbody>
</table>

The trajectory of the module at Descent orbit insertion (Engine cutoff) and Powered descent initiation (Engine ignition) is provided in the *Apollo 11 Mission Report* (Table 7-II.- Trajectory Parameters).

<table>
<thead>
<tr>
<th>Var1</th>
<th>Latitude_deg</th>
<th>Longitude_deg</th>
<th>Altitude_mi</th>
<th>Altitude_ft</th>
<th>Velocity_fps</th>
</tr>
</thead>
<tbody>
<tr>
<td>{'Descent Orbit Insertion (Engine cutoff)'}</td>
<td>-1.16</td>
<td>-141.88</td>
<td>57.8</td>
<td>3.512e+05</td>
<td>5284.9</td>
</tr>
<tr>
<td>{'Powered Descent (Engine ignition)'}</td>
<td>1.02</td>
<td>39.39</td>
<td>6.4</td>
<td>38887</td>
<td>5564.9</td>
</tr>
</tbody>
</table>

The Simulink model aero_dap3dof is initialized to align with the beginning of the mission phase "Descent Orbit Insertion trajectory (Engine cutoff)". Open the aero_dap3dof model and run the simulation.

**Closing Remarks**

Building a digital autopilot was a daunting task in 1961 because there was very little industrial infrastructure for it - everything about it was in the process of being invented. Here is an excerpt from the technical article *The Apollo 11 Moon Landing: Spacecraft Design Then and Now*:

"One reason why the [autopilot's machine code] was so complex is that the number of jets that could be used to control the rotations about the pilot axes was large. A decision was made to change the axes that the autopilot was controlling to the "jet axes" shown in aero_dap3dof. This change dramatically reduced the number of lines of code and made it much easier to program the autopilot in the existing computer. Without this improvement, it would have been impossible to have the autopilot use only 2000 words of storage. The lesson of this change is that when engineers are given the opportunity to code the computer with the system they are designing, they can often modify the design to greatly improve the code."
References


Designing a High Angle of Attack Pitch Mode Control

Control Design Using Simulink®

This example shows how to use the Control System Toolbox™ and Simulink® Control Design™ to interact with Simulink to design a digital pitch control for the aircraft. In this example, we will design the controller to permit the aircraft to operate at a high angle of attack with minimal pilot workload.

Our example takes you through the first pass at designing a digital autopilot for a high angle of attack controller. To run everything in this example you must have the Control System Toolbox, Simulink Control Design, Simulink, and Simulink® Coder™. If you don't have all of these products, you can still run portions of the example using cell execution mode of the MATLAB® editor.

Below is a Simulink model of the aircraft. The control systems in the Controllers block can be switched in the model to allow you to see the analog response and then to switch to a design created using the Control System Toolbox's Linear Time Invariant (LTI) objects. A controller is also included that is a discrete implementation of the analog design that is similar to the algorithm that would go into an on-board flight computer. Take a few moments to explore the model.

Open the aircraft model
Aircraft Digital Flight Control System - High Angle of Attack Mode

Pilot

Stick Command

Controllers

Digital Design
(Ilti or Implementation)

Dryden Wind Gust

wdot gust

qdot gust

Aircraft Dynamic Model

Alpha

q

Linear Actuator Subsystem

Act. Pos.


Variant Actuator

Actuator

Analog or Digital

Analog

lti

DAP

Copyright 2013-2014 The MathWorks, Inc.
Figure 1: Simulink model of the aircraft flight control system.

**Trim and Linearization**

The model can be linearized using the Linear Analysis Tool, launched from `slexAircraftPitchControlAutopilot` Simulink model. In the Simulink Editor, on the **Apps** tab, under **Control Systems**, click **Model Linearizer**.
For help type help slcontrol or help ltiview or look at the Control System Toolbox and Simulink Control Design product documentation.

Open the slexAircraftPitchControlAutopilot model

To view the linearized model parameters:

```matlab
apmdl = 'slexAircraftPitchControlAutopilot';
open_system(apmdl);
op = operpoint(apmdl);
io = getlinio(apmdl);
contap = linearize(apmdl,op,iop)
```

Divide the model into its A, B, C, and D matrices:

```matlab
contap =

A =
    Alpha-sensor     Pitch Rate L   Proportional   Stick Prefil
Alpha-sensor              -2.526             0             0             0
Pitch Rate L                        0        -4.144             0             0
Proportional                     -1.71        0.9567             0            10
Stick Prefil                                    0             0           -10

B =
    Alpha Sensed    Stick    q Sensed
Alpha-sensor                1             0             0
Pitch Rate L                                0            1
Proportional                              0             0            -0.8156
Stick Prefil                              0            1             0

C =
    Alpha-sensor     Pitch Rate L   Proportional   Stick Prefil
Sum                   2.986         -1.67        -3.864           -17.46

D =
    Alpha Sensed    Stick    q Sensed
Sum                0             0         1.424

Continuous-time state-space model.
The Original Analog Autopilot for the High Angle of Attack Mode

**Figure 2:** Original analog autopilot.

**Linear Time-Invariant (LTI) Systems**

There are three types of LTI objects you can use to develop a linear model:

State Space (SS), Transfer Function (TF), and Zero-Pole-Gain (ZPG) objects.

The variable contap is a State Space object. You can then get one of the other types with the other commands. When you create the object in MATLAB, you can manipulate it using operations such as *, +, -, etc. This is called "overloading" the MATLAB operators. Try creating an object of your own and see what happens when adding, multiplying, etc. with the contap object.
To see exactly what is stored in the LTI object, type `get(contap)` or `contap.InputName` for example.

To view Zero/Pole/Gain transfer functions:

```matlab
ccontap = tf(contap);
ccontap = zpk(contap)
```

\[
\text{contap} = \begin{align*}
\text{From input "Alpha Sensed" to output "Sum":} & \quad 2.9857 \ \frac{(s+2.213)}{s (s+2.526)} \\
\text{From input "Stick" to output "Sum":} & \quad -17.46 \ \frac{(s+2.213)}{s (s+10)} \\
\text{From input "q Sensed" to output "Sum":} & \quad 1.424 \ \frac{(s+2.971) (s+2.213)}{s (s+4.144)}
\end{align*}
\]

Continuous-time zero/pole/gain model.

**Discretized Controller Using Zero-Order Hold**

Now the LTI object will be used to design the digital autopilot that will replace the analog autopilot. The analog system is coded into the LTI object called `contap` (CONtinuous AutoPilot).

The first attempt at creating a digital autopilot will use a zero-order hold with a sample time of 0.1 seconds. Note that the discrete object maintains the type (ss, tf, or zpk).

It is clear from Bode plot below that the systems do not match in phase from 3 rad/sec to the half sample frequency (the vertical black line) for the pilot stick input and the angle of attack sensor. This design has poorer response than the analog system. Go to the Simulink model and start the simulation (make sure you can see the scope windows). While the simulation is running, double-click the manual switch labeled Analog or Digital.
Does the simulation verify the conclusion reached by interpreting the Bode diagram?

discap = c2d(contap, 0.1, 'zoh');
get(discap)
bode(contap,discap)

Z: {
    [0.8016]  [0.8039]  [2x1 double]
}
P: {
    [2x1 double]  [2x1 double]  [2x1 double]
}
K: [0.2943 -1.2458 1.4240]
DisplayFormat: 'roots'
Variable: 'z'
IODelay: [0 0 0]
InputDelay: [3x1 double]
OutputDelay: 0
Ts: 0.1000
TimeUnit: 'seconds'
InputName: {3x1 cell}
InputUnit: {3x1 cell}
InputGroup: [1x1 struct]
OutputName: {'Sum'}
OutputUnit: {''}
OutputGroup: [1x1 struct]
Notes: [0x1 string]
UserData: []
Name: ''
SamplingGrid: [1x1 struct]
**Tustin (Bilinear) Discretization**

Now try different conversion techniques. You can use the Tustin transformation. In the command window type the commands above.

It should be clear that the systems still do not match in phase from 3 rad/sec to the half sample frequency, the Tustin transformation does better. The simulation uses the LTI object as it is designed. To see how the object is used look in the Controllers subsystem by using the browser or by double clicking the icon. The LTI block picks up an LTI object from the workspace. You can change the object name used in the block to any LTI object in the workspace.

**Figure 3:** Bode diagram comparing analog and ZOH controllers.
Try using "discap1", the Tustin discretization of the analog design:

```matlab
discap1 = c2d(contap,0.1,'tustin');
bode(contap,discap,discap1)
```

**Figure 4:** Bode diagram comparing analog and 0.1 sec Tustin controllers.

**Selecting a Sample Time**

The Tustin transform performs better than the zero-order hold from the analysis so far. The sample time of 0.1 second appears to be too slow for the discrete system to track the performance of the analog system at half the sample frequency.
Now transform the analog design using the Tustin transform with a 0.05 second sample period:

discap = c2d(contap,0.05,'tustin')
bode(contap,discap)

discap =

From input "Alpha Sensed" to output "Sum":
0.074094 (z-0.8951) (z+1)  
--------------------------
(z-1) (z-0.8812)

From input "Stick" to output "Sum":
-0.36852 (z-0.8951) (z+1)  
---------------------------
(z-1) (z-0.6)

From input "q Sensed" to output "Sum":
1.4629 (z-0.8617) (z-0.8951)  
-----------------------------
(z-1) (z-0.8123)

Sample time: 0.05 seconds
Discrete-time zero/pole/gain model.
Figure 5: Bode diagram comparing analog and 0.05 sec Tustin controllers.

Real-World Considerations

Now that we have what appears to be a workable design, we need to implement it in a form that will include some of the necessary elements that were ignored in the linear analysis. For example, if you look at the analog autopilot that is in the Controllers subsystem, you will see some logic that stops the integrator from winding up when the actuator saturates:

Open the AnalogControl subsystem
Implementation of the Full Design

The integrator wind-up is only one of the practical issues that needs to be addressed. Another is the need to eliminate aliased high frequency signals that could enter at measurement points. These aliased signals can be prevented by specifying analog filters in front of the sample and hold (ZOH) blocks that model the analog-to-digital converter (ADC) devices typically used for measuring signals in an embedded application. Additionally, the filters in the digital section of the autopilot operate at a higher sampling rate than the compensator to provide a conditioned signal to the compensator. The analog filters are necessary because once a signal is aliased, there is no way to separate the valid in-band frequency content from aliased content.

This new controller is called Digital Control. The filters have a sample time of deltat1 (set to be 1/10 of deltat). The zero-order hold blocks specify these sample times for various downstream blocks through sample time inheritance.

Figure 6: AnalogControl subsystem.
The switches in the slexAircraftPitchControlExample model are set up so you can switch between the analog autopilot, the digital LTI object, and the digital autopilot that could be implemented using software.

Try simulating the system again and switch among the three autopilot designs. You should see that the designs are not significantly affected by which autopilot is active. You can also increase the amplitude of the wind gust and verify that the anti-aliasing filters are working satisfactorily. To increase the gust amplitude, open the Dryden Wind Gust subsystem and change (by double clicking the icon) the noise variance of the White Noise that drives the gust simulation.

Open the DigitalControl subsystem:

![DigitalControl subsystem diagram]

**Figure 7:** DigitalControl subsystem.

**Variant Systems for Design Variants**

The actuators in the Simulink model use the Variant Subsystems block to represent multiple actuator implementations, where only one implementation is active during simulation. You can double-click the Variant Actuator Subsystem block to view the different actuator implementations.
To make the simulation change, use the nonlinear actuator instead of the linear one. You can do this by using the Variant Subsystem block's "Variant > Override using" context menu.

Note that Simulink must be stopped in order to reconfigure the actuator selection. You should also note that the nonlinear actuator has saturations on position and rate.

Figure 8: Variant Subsystem for Actuators

Code Generation

The autopilot design can be transformed into embeddable code using Simulink Coder. A separate model of the digital autopilot, slexAircraftPitchControlDAP, is open below; it was coded into a host-based standalone program using Simulink Coder and the compiler specified using the mex -setup configuration. The simulation results can be used as a validation baseline for the functional correctness of the generated code.

The executable file resulting from code generation and compilation is slexAircraftPitchControlDAP.exe. If you create it, it can be run from the OS shell (DOS) command line directly or from MATLAB by typing:

$$ >> !\text{slexAircraftPitchControlDAP.exe} $$

The executable program creates a file called slexAircraftPitchControlDAP.mat that you can load by typing load slexAircraftPitchControlDAP.

Two variables called rt_tout and rt_yout will be in the workspace and their variance against the simulation baseline can be plotted. Note that the variance is effectively zero.
for this model on this host. Some small numeric variance is to be expected in more complex calculations due to differing compiler optimizations and use of intermediate register variables having higher precision than the 64-bit storage format of a double. Significant variances should be examined as they could indicate a numerical stability problem in your model's algorithms, a compiler or run-time library bug, or other problems.

Open the slexAircraftPitchControlDAP model:
**Figure 9:** Comparison of simulation and code generation results.

**Behavior of the First Pass Design**

Here is the time response of the high angle of attack mode digital pitch controller design:
**Summary**

Further work on this design might include incorporating and analyzing more of the real-world effects on the compensator such as the effect of the anti-aliasing filter dynamics and computational delay of the embedded digital computer on the overall "plant" seen by the digital computations. The combination of the real plant, the computational delay, the anti-aliasing filters, and the sample-and-hold delay are just a few of the additional items that might affect the design of the digital compensator parameters.

---

**Figure 10:** Response of the aircraft Digital Pitch Control Design.
Six Degrees of Freedom (6-DoF) Motion Platform

This example shows how to model six degrees of freedom motion in Simulink®. You can switch between using Euler Angles and Quaternions to model the equations of motion, using the Variant Subsystem block's "Variant > Override using" context menu.
Six Degrees of Freedom (6-DoF) Motion Platform
Aircraft Longitudinal Flight Control

This example shows how to model flight control for the longitudinal motion of an aircraft. First order linear approximations of the aircraft and actuator behavior are connected to an analog flight control design that uses the pilot's stick pitch command as the set point for the aircraft's pitch attitude and uses aircraft pitch angle and pitch rate to determine commands. A simplified Dryden wind gust model is incorporated to perturb the system.

The Viewers and Generators Manager was used in this model to create the model scope for viewing signals, Scope1. Found on Simulation -> Prepare -> Viewers Manager, the Viewers and Generators Manager allows you to instrument your model without having to add blocks to the model. If you close a model scope, such as for batch simulations, you can later reopen it by double-clicking the "scope" icon on the signal of interest.

Parameters for the model are stored in a file named slexAircraftData.m. This file is loaded by the model into the model workspace. You can view and edit data in the model workspace directly by using the Model Explorer, which is launched using the Modeling -> Design -> Model Explorer menu item.
Aircraft Longitudinal Flight Control

This example models a flight control algorithm for the longitudinal motion of an aircraft.

Copyright 2012-2019 The MathWorks, Inc.
Radar Tracking Using MATLAB Function Block

This example shows how to use an extended Kalman filter with the MATLAB® Function block in Simulink® to estimate an aircraft's position from radar measurements. The filter implementation is found in the MATLAB Function block, the contents of which are stored in the Simulink model itself.

At the end of the simulation, three figures display the following information: the actual trajectory compared to the estimated trajectory; the estimation residual for range; and the actual, measured, and estimated positions.
Optical Sensor Image Generation

Generating an Optical Sensor Image From Simulated Movement Data

This example shows how to generate a movie with 64 frames and a frame size of 64 by 64 pixels (at 10 frames per second). The movie contains a simulation of a moving target that is moving through a structured background that is itself moving. A jitter motion caused by random vibration is also generated (in a Simulink® model called "aero_vibrati") and the jitter motion is added into the overall sensor motion. Finally, the image is blurred through a Gaussian optical point spread function.

Note: Changing delt here also requires a change in the parameters set-up dialog box in the Simulink model "vibration".

delt = 0.1;      % Sample time of the generated sequence
num_frames= 64;  % Number of frames to generate
framesize = 64;  % Square frame size in pixels

out = zeros(framesize,framesize,num_frames);    % Initialize movie storage as a 3D Array

Generate a Target and Define Its Motion

The first stage is to define the shape and motion of the target object. The shape chosen is a large plus sign, and the image is defined by a matrix representing the image intensity at each pixel position. The Target is defined to be traveling from center to bottom right of the image.

target = [zeros(3,11)
          zeros(1,5) 6 zeros(1,5)
          zeros(1,5) 6 zeros(1,5)
          zeros(1,3) 6 6 6 6 zeros(1,3)
          zeros(1,5) 6 zeros(1,5)
          zeros(1,5) 6 zeros(1,5)
          zeros(1,3) 6 6 6 6 zeros(1,3)]; % Target is a plus sign 5 by 5 pixels across

 target_velx = 1;                    % target velocity in x direction in pixels per second
 target_vely = 1;                    % target velocity in y direction in pixels per second
 target_x_initially = framesize/2;   % the target is initially in the center of the frame in x
 target_y_initially = framesize/2;   % and in y

figure(1);
colormap('gray');
image(target*32);
title('Target Image')
Build Background and Target Composite Image

Generate a sinusoidally correlated background and give it a drift motion. Then, overlay the target onto the background image.

backsize = framesize+36;  % Make the background bigger than the frame so when it drifts there are new pixels available to drift into.
xygrid = (1:backsize)/backsize;
B=2*sin(2*pi*xygrid).^2'*cos(2*pi*xygrid).^2;
psd = fft2(B);
psd = real(psd.*conj(psd));

background = B + 0.5*randn(backsize);    % Add a specular Gaussian white noise.
% sequence to the structure with
% variance of 0.25 (sigma of 0.5).

xoff = 10;
yoff = 10;  % Sensor location is offset from the 0,0 of the background
driftx = 1;
drifty = 1;  % drift rate of the background in a and y directions pix/sec.

minout = min(min(min(background))); 
maxout = max(max(max(background))); 
colormap('gray');
image((background-minout)*64/(maxout-minout))
title('Background image with additive white specular noise')
Simulate the Tracker's Rotational Vibration

Rotational vibration of the tracker is simulated using model aero_vibrati. The data required to simulate the vibration of the tracker is generated by running the Simulink model "aero_vibrati".

Run Simulink vibration model using sim command (Note -- if the delt is changed from 0.1 seconds, the Simulink model must be changed also to ensure that the sample time for the vibration match the sample time in this tracker image model.

The resulting random rotations are shown in Figure 1.

\[
\begin{align*}
\omega &= 2\pi \times 5; \quad \text{% The structural frequencies are 5, 10 and 15 Hz in the model.} \\
zeta &= 0.01; \quad \text{% Damping ratio for all modes}
\end{align*}
\]

\[
\text{open_system('aero_vibrati')} \\
\text{simout = sim('aero_vibrati','SrcWorkspace','current');}
\]

\[
\begin{align*}
vibdat &= \text{simout.get('vibdat');} \quad \text{% The Simulink model "aero_vibrati"}
&\quad \text{generates the vibration data at}
&\quad \text{a sample time of 0.01 sec.}
\vibx &= \text{vibdat(1:10:1000);} \quad \text{% The output of simulation is}
&\quad \text{returned as the variable simout}
&\quad \text{The variable simout contains}
\viby &= \text{vibdat(1001:10:2000);} \quad \text{% the in array vibdat that contains}
&\quad \text{the vibration data}
\end{align*}
\]

\[
\begin{align*}
\text{levarmx} &= 10; \quad \text{% Rotational lever arm for vibration noise in x}
\text{levarmy} &= 10; \quad \text{% and in y.}
\end{align*}
\]

\[
\text{subplot(211);}
\text{plot(0.01*(1:10:1000),vibx);grid;}
\text{title('Time history of the random Tracker rotations')}
\text{xlabel('Time');ylabel('x direction')}
\]

\[
\text{subplot(212);}
\text{plot(0.01*(1:10:1000),viby);grid;}
\text{xlabel('Time');ylabel('y direction')}
\]
Simulate the Motion Effects From the Background, Target, and Jitter

The frames that will make up the movie are now created and stored in a multidimensional array (out). Each frame has the background and target at differing positions due to the target motion, background drift, and tracker vibration. The first frame of the movie will be shown in Figure 1.

clf; drawnow;

for t = 1:num_frames

  % Drift the Background at the rate driftx and drifty
  % (in pixels/second) and add in the vibration:
  xshift = driftx*delt*t+levarmx*vibx(t,1);
yshift = drifty*delt*t+levarmy*viby(t,1);

% Interpolate the 2D image using the MATLAB(R) function interp2:
[xgrid, ygrid] = meshgrid(1:backsize);
[xindex, yindex] = meshgrid(xshift:1:xshift+backsize,yshift:1:yshift+backsize);
outtemp = interp2(xgrid,ygrid,background,xindex,yindex);

% Truncate the drifted image down from backsize to framesize:
out(:,:,t) = outtemp(xoff:xoff+framesize-1,xoff:xoff+framesize-1);

% Now let the target move also:
tpixinx = floor(target_velx*delt*t);
tpixiny = floor(target_vely*delt*t);  % Before interpolating extract the number of pixels moved
txi = target_velx*delt*t - tpixinx;
tyi = target_vely*delt*t - tpixiny;   % Interpolate on sub-pixels around the origin only
[txi tyi] = meshgrid(txi+1:txi+11,tyi+1:tyi+11); % meshgrid generates 2 matrices with the x and y grids

% Interpolate the intensity values first using interp2 -- a built in MATLAB command
temp = interp2(txgrid,tygrid,target,txi,tyi);

% Insert the target at the location determined by the initial offset, and the number of inter-pixels moved
out(tx:tx+6,ty:ty+6,t) = temp(9:-1:3,9:-1:3) + out(tx:tx+6,ty:ty+6,t);

end

minout = min(min(min(out)));  
maxout = max(max(max(out)));  
colormap('gray');  
image((out(:,:,1)-minout) * 64/(maxout-minout));  
title('First frame of combined target and background image.')
Pass the Images Through Optics -- Use a Gaussian "Aperture Function"

This code segment can use a measured aperture function just as easily - simply replace the next five lines by "load measured_aperture" where measured_aperture is the measured function stored in ASCII and the data stored in the file measured_aperture.mat is a MATLAB®.mat file that contains the matrix apfunction. (in MATLAB type "help load" for how to use load and look at the c and fortran code that shows how to read and write MATLAB .mat files).

(Note: When the Point Spread Function is Gaussian, then so is the Aperture function)
To simulate the effect of the tracker optics, each of the movie frames is now blurred using a 2-D FFT (Fast Fourier Transform). The first frame of the resulting image is shown in Figure 1.

```matlab
x = 1:framesize;
y = 1:framesize;
sigma = 120;
apfunction = exp(-(x-framesize/2).^2/(2*sigma))' * exp(-(y-framesize/2).^2/(2*sigma));
apfunction = fftshift(apfunction); % Rotate so it conforms with FFT convention

for j = 1:num_frames
    out(:,:,j) = real(ifft2(apfunction.*fft2(out(:,:,j))));
end

minout = min(min(min(out)));
maxout = max(max(max(out)));
colormap('gray');
image((out(:,:,1)-minout)*64/(maxout-minout));
title('First frame of blurred image.')```
Generate the MATLAB® Movie and Play It Back

Scale the movie frame so that it has 64 intensity values from the min to the max and then show the result as an image. See MATLAB help for how the moviein and getframe work.

\[
\text{minout} = \min(\min(\min(\text{out}))); \\
\text{maxout} = \max(\max(\max(\text{out}))); \\
\]

\[
\text{M} = \text{moviein}(\text{num\_frames}) ; \\
\text{for } j = 1 : \text{num\_frames} \\
\quad \text{image}((\text{out}(:,:,j) - \text{minout}) * 64 / (\text{maxout} - \text{minout})) \\
\quad \text{drawnow} \\
\quad \text{M}(:,:,j) = \text{getframe}; \\
\text{end}
\]
**OPTIONAL: Save the Movie in a .mat File**

You can optionally save the generated tracker movie in a mat file and also save the psd of the background for later use with the movie.

```matlab
save trackerimage out
save psdback psd
save moviedat M
```
bdclose('aero_vibrati');
Air Traffic Control Radar Design

This example shows how to model a conceptual air traffic control (ATC) radar simulation based on the radar range equation.

**Model Description**

To make parameters for radar system design easier to change and easier to determine their values, a GUI is supplied with this model. Radar and weather parameters may be changed from this GUI. While simulating, the effects of these parameters can be seen on the scope display which shows the actual aircraft range in yellow and the estimated aircraft range from the radar in magenta. Another output that can be viewed is the calculated signal to noise ratio (SNR) is compared to the ideal SNR. Ideal SNR is also specified from the GUI. The result is shown in the display block and will be either 1 (SNR $\geq$ ideal SNR) or 0 (SNR $<$ ideal SNR).

Simulink® and Stateflow® are used in the model, which is divided into three main subsystems, radar, aircraft, and weather.

Using subsystems is helpful in two ways: the model is organized and easier to understand and the work can be split between multiple engineers by subsystems. The Stateflow machine labeled "check SNR" performs the logic comparing calculated SNR to the ideal SNR and output data based on this comparison.

You can run the simulation to determine if the radar can pick up the aircraft by the output on the scope. Using the GUI, the radar and the weather parameters can be altered and will change the range where the aircraft can be "seen".

Open the aero_atc model

```matlab
open_system('aero_atc');
sim('aero_atc');
```
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>RADAR Band</td>
<td>L Band</td>
</tr>
<tr>
<td>Wavelength (m)</td>
<td>0.15</td>
</tr>
<tr>
<td></td>
<td>0.285</td>
</tr>
<tr>
<td></td>
<td>0.3</td>
</tr>
<tr>
<td>Transmitter Power (kW)</td>
<td>50</td>
</tr>
<tr>
<td>Losses (dB)</td>
<td>5</td>
</tr>
<tr>
<td>Noise Factor (dB)</td>
<td>5</td>
</tr>
<tr>
<td>Antenna Efficiency</td>
<td>0.7</td>
</tr>
<tr>
<td>RADAR Range (N. Mile)</td>
<td>50</td>
</tr>
<tr>
<td>Elev. Separation (N. Mile)</td>
<td>2</td>
</tr>
<tr>
<td>Lat. Separation (N. Mile)</td>
<td>2</td>
</tr>
<tr>
<td>Range Resolution (m)</td>
<td>150</td>
</tr>
<tr>
<td>Bandwidth (Hz)/Pulsewidth</td>
<td>1.2</td>
</tr>
<tr>
<td>Reliable Detection S/N (dB)</td>
<td>15</td>
</tr>
<tr>
<td>Working Temperature (deg F)</td>
<td>15</td>
</tr>
<tr>
<td>Weather Condition</td>
<td></td>
</tr>
<tr>
<td>No Precipitation</td>
<td></td>
</tr>
<tr>
<td>Drizzle</td>
<td></td>
</tr>
<tr>
<td>Light Precipitation</td>
<td></td>
</tr>
<tr>
<td>Moderate Precipitation</td>
<td></td>
</tr>
<tr>
<td>Heavy Precipitation</td>
<td></td>
</tr>
</tbody>
</table>
Radar systems are designed for a specific purpose and can very seldom be used for other applications effectively. Each new radar specification requires the computation of new parameter values. When designing a radar for an application, there are a number of parameters which shape the design. Some of these parameters are contained or derived logically from the customer specification. Others are selected arbitrarily using the design engineer's best judgment. This is the first approximate solution for the system design. From here, continual refinement of the design parameters takes place until an optimum design is reached. If any changes occur in the customer specification, it could cause a need to rework the design process over from the beginning. The parametric nature of this design strategy lends itself to automation.
We're interested in performing conceptual design for a ground-based air traffic control (ATC) radar. Let's take a look at a potential customer specification.
This is an example of a customer specification upon which a design process would be based. The customer, possibly the FAA, provides some basic requirements for the radar design leaving a number of parameter selections up to the design engineer.

It should be noted that some of the logically derived parameters are dependent on assumptions made by the engineer and would need to be re-calculated each time the best-judgment parameters are optimized. This problem lends itself well to simulation. By using Simulink and Stateflow, the design engineer has the analysis capability to have time-varying design cases for Monte Carlo test runs, i.e.: aircraft cross-sections and locations, weather cross-sections, and locations.

**MathWorks® Products in the Design Process**

Here's how MathWorks® products fit the job of conceptual radar design:
Using the customer specification and the radar range equations along with equations describing the physics of the system, a model is built in MATLAB®, Simulink, and Stateflow. Using the model with the sim command for batch runs, those best-judgment parameters can be optimized for various conditions, weather, aircraft, using a Monte Carlo simulation run to prove robustness. The result is a set of optimized radar parameters that can be used to build a detailed block diagram model of the full radar system for further system analysis in Simulink with the DSP System Toolbox™.
Designing a Guidance System in MATLAB and Simulink

This example shows how to use the model of the missile airframe presented in a number of published papers (References [1], [2] and [3]) on the use of advanced control methods applied to missile autopilot design. The model represents a tail controlled missile travelling between Mach 2 and Mach 4, at altitudes ranging between 10,000ft (3,050m) and 60,000ft (18,290m), and with typical angles of attack ranging between +/-20 degrees.

Model of the Airframe Dynamics

The core element of the model is a nonlinear representation of the rigid body dynamics of the airframe. The aerodynamic forces and moments acting on the missile body are generated from coefficients that are non-linear functions of both incidence and Mach number. The model can be created with Simulink® and the Aerospace Blockset™. The aim of this blockset is to provide reference components, such as atmosphere models, which will be common to all models irrespective of the airframe configuration. Simplified versions of the components available in the Aerospace Blockset are included with these examples to give you a sense of the potential for reuse available from standard block libraries.
Open the model

**Missile Guidance System Example**

![Diagram of Missile Guidance System]

Representing the Airframe in Simulink

The airframe model consists of four principal subsystems, controlled through the acceleration-demand autopilot. The Atmosphere model calculates the change in atmospheric conditions with changing altitude, the Fin Actuator and Sensors models couple the autopilot to the airframe, and the Aerodynamics and Equations of Motion model calculates the magnitude of the forces and moments acting on the missile body, and integrates the equations of motion.
International Standard Atmosphere Model

\[
\begin{align*}
T &= T_0 - Lh \\
\rho &= \rho_0 \left( \frac{T}{T_0} \right)^{\left( \frac{g}{L} \right) - 1} \\
P &= P_0 \left( \frac{T}{T_0} \right)^{\left( \frac{g}{L} \right)} \\
a &= \sqrt{\gamma RT}
\end{align*}
\]

Troposphere Model

\( T_0 \) = Absolute temperature at mean sea level [K]
\( \rho_0 \) = Air density at mean sea level [Kg/m^3]
\( P_0 \) = Static pressure at mean sea level [N/m^2]
\( h \) = Altitude [m]
\( T \) = Temperature at altitude \( h \)
\( \rho \) = Air density at altitude \( h \)
\( P \) = Static pressure at altitude \( h \)
\( a \) = Speed of sound at altitude \( h \)
\( L \) = Lapse rate [K/m]
\( \gamma \) = Characteristic gas constant [J/Kg/K]
\( g \) = Acceleration due to gravity [m/s^2]

The Atmosphere Subsystem that is used is an approximation to the International Standard Atmosphere, and is split into two separate regions. The troposphere region lies between sea level and 11Km, and in this region there is assumed to be a linear temperature drop with changing altitude. Above the troposphere lies the lower stratosphere region ranging between 11Km and 20Km. In this region the temperature is assumed to remain constant.
Aerodynamic Coefficients for Constructing Forces and Moments

The Aerodynamics & Equations of Motion Subsystem generates the forces and moments applied to the missile in body axes, and integrates the equations of motion which define the linear and angular motion of the airframe.

The aerodynamic coefficients are stored in datasets, and during the simulation the value at the current operating condition is determined by interpolation using 2-D lookup table blocks.
Classical Three Loop Autopilot Design

The aim of the missile autopilot is to control acceleration normal to the missile body. In this example the autopilot structure is a three loop design using measurements from an accelerometer placed ahead of the center of gravity, and a rate gyro to provide additional damping. The controller gains are scheduled on incidence and Mach number, and are tuned for robust performance at an altitude of 10,000 ft.

To design the autopilot using classical design techniques requires that linear models of the airframe pitch dynamics be derived about a number of trimmed flight conditions. MATLAB® can determine the trim conditions, and derive linear state space models directly from the non-linear Simulink model, saving both time, and aiding in the validation of the model that has been created. The functions provided by the MATLAB Control System Toolbox™ and Simulink® Control Design™ allow the designer to visualize the behavior of the airframe open loop frequency (or time) responses. To see how to trim and linearize the airframe model you can run the companion example, "Airframe Trim and Linearize".
Airframe Frequency Response

Autopilot designs are carried out on a number of linear airframe models derived at varying flight conditions across the expected flight envelope. To implement the autopilot in the non-linear model involves storing the autopilot gains in 2 dimensional lookup tables, and incorporating an anti-windup gain to prevent integrator windup when the fin demands exceed the maximum limits. Testing the autopilot in the nonlinear Simulink model is then the best way to show satisfactory performance in the presence of non-linearities such as actuator fin and rate limits, and with the gains now dynamically varying with changing flight condition.
**Figure**: Simulink implementation of gain scheduled autopilot

**Homing Guidance Loop**

The complete Homing Guidance Loop consists of a Seeker/Tracker Subsystem which returns measurements of the relative motion between the missile and target, and the Guidance Subsystem which generates normal acceleration demands which are passed to the autopilot. The autopilot is now part of an inner loop within the overall homing guidance system. Reference [4] provides information on the differing forms of guidance that are currently in use, and provides background information on the analysis techniques that are used to quantify guidance loop performance.
Guidance Subsystem

The function of the Guidance subsystem is to not only generate demands during closed loop tracking, but also perform an initial search to locate the target position. A Stateflow® model is used to control the transfer between these differing modes of operation. Switching between modes is triggered by events generated either in Simulink, or internal to the Stateflow model. Controlling the way the Simulink model then behaves is achieved by changing the value of the variable **Mode** that is passed out to Simulink. This variable is used to switch between the differing control demands that can be generated. During target search the Stateflow model controls the tracker directly by sending demands to the seeker gimbals (**Sigma**). Target acquisition is flagged by the tracker once the target lies within the beamwidth of the seeker (**Acquire**), and after a short delay closed loop guidance starts. Stateflow is an ideal tool for rapidly defining all the operational modes, whether they are for normal operation, or unusual situations. For example, the actions to be taken should there be loss of lock on the target, or should a target not be acquired during target search are catered for in this Stateflow diagram.
Proportional Navigation Guidance

Once the seeker has acquired the target a Proportional Navigation Guidance (PNG) law is used to guide the missile until impact. This form of guidance law has been used in guided missiles since the 1950s, and can be applied to radar, infrared or television guided missiles. The navigation law requires measurements of the closing velocity between the missile and target, which for a radar guided missile could be obtained using a Doppler tracking device, and an estimate for the rate of change of the inertial sightline angle.
The aim of the Seeker/Tracker Subsystem is both to drive the seeker gimbals to keep the seeker dish aligned with the target, and to provide the guidance law with an estimate of the sightline rate. The tracker loop time constant $\tau_{ors}$ is set to 0.05 seconds, and is chosen as a compromise between maximizing speed of response, and keeping the noise transmission to within acceptable levels. The stabilization loop aims to compensate for body rotation rates, and the gain $K_s$, which is the loop cross-over frequency, is set as high as possible subject to the limitations of the bandwidth of the stabilizing rate gyro. The sightline rate estimate is a filtered value of the sum of the rate of change of the dish angle measured by the stabilizing rate gyro, and an estimated value for the rate of change of
the angular tracking error (e) measured by the receiver. In this example the bandwidth of the estimator filter is set to half that of the bandwidth of the autopilot.

Radome Aberration

For radar guided missiles a parasitic feedback effect that is commonly modelled is that of radome aberration. It occurs because the shape of the protective covering over the seeker distorts the returning signal, and then gives a false reading of the look angle to the target. Generally the amount of distortion is a nonlinear function of the current gimbal angle, but a commonly used approximation is to assume a linear relationship between the gimbal angle and the magnitude of the distortion. In the above system, the radome aberration is accounted for in the gain block labeled "Radome Aberration". Other parasitic effects, such as sensitivity in the rate gyros to normal acceleration, are also often modelled to test the robustness of the target tracker and estimator filters.
**Figure**: Radome aberration geometry

**Running the Guidance Simulation**

Now to show the performance of the overall system. In this case the target is defined to be travelling at a constant speed of 328m/s, on a reciprocal course to the initial missile heading, and 500m above the initial missile position. From the simulation results it can be determined that acquisition occurred 0.69 seconds into the engagement, with closed loop guidance starting after 0.89 seconds. Impact with the target occurred at 3.46 seconds, and the range to go at the point of closest approach was calculated to be 0.265m.

The `aero_guid_plot.m` script creates a performance analysis
Missile Guidance System Example

Copyright 1990-2014 The MathWorks, Inc.
The animation block provides a visual reference for the simulation.
References


Airframe Trim and Linearize

This example shows how to trim and linearize an airframe using Simulink® Control Design™

Designing an autopilot using classical design techniques requires linear models of the airframe pitch dynamics for a number of trimmed flight conditions. MATLAB® can determine the trim conditions and derive linear state-space models directly from the nonlinear Simulink® and Aerospace Blockset™ model. This saves time and helps to validate the model. The functions provided by Simulink Control Design allow you to visualize the behavior of the airframe in terms of open-loop frequency (or time) responses.

Initialize Guidance Model

The first problem is to find the elevator deflection, and the resulting trimmed body rate (q), which will generate a given incidence value when the missile is travelling at a set speed. Once the trim condition is found, a linear model can be derived for the dynamics of the perturbations in the states around the trim condition.

```
open_system('aero_guidance_airframe');
```
Define State Values

\[
\begin{align*}
    h_{ini} &= 10000/m2ft; \quad \% \text{Trim Height [m]} \\
    M_{ini} &= 3; \quad \% \text{Trim Mach Number} \\
    \alpha_{ini} &= -10*d2r; \quad \% \text{Trim Incidence [rad]} \\
    \theta_{ini} &= 0*d2r; \quad \% \text{Trim Flightpath Angle [rad]}
\end{align*}
\]
\[ v_{ini} = M_{ini} \times (340 + (295 - 340) \times h_{ini}/11000); \quad \% \text{Total Velocity [m/s]} \]

\[ q_{ini} = 0; \quad \% \text{Initial pitch Body Rate [rad/sec]} \]

**Set Operating Point and State Specifications**

The first state specifications are Position states, the second state specification is Theta. Both are known, but not at steady state. The third state specifications are body axis angular rates of which the variable w is at steady state.

\[
\begin{align*}
\text{opspec} &= \text{operspec('aero_guidance_airframe');} \\
\text{opspec.State}(1).\text{Known} &= [1;1]; \\
\text{opspec.State}(1).\text{SteadyState} &= [0;0]; \\
\text{opspec.State}(2).\text{Known} &= 1; \\
\text{opspec.State}(2).\text{SteadyState} &= 0; \\
\text{opspec.State}(3).\text{Known} &= [1\ 1]; \\
\text{opspec.State}(3).\text{SteadyState} &= [0\ 1];
\end{align*}
\]

**Search for Operating Point, Set I/O, then Linearize**

\[
\begin{align*}
\text{op} &= \text{findop('aero_guidance_airframe',opspec);} \\
\text{io}(1) &= \text{linio('aero_guidance_airframe/Fin Deflection',1,'input');} \\
\text{io}(2) &= \text{linio('aero_guidance_airframe/Selector',1,'output');} \\
\text{io}(3) &= \text{linio(sprintf(['aero_guidance_airframe/Aerodynamics &
\text{Equations of Motion'}]),3,'output');}
\end{align*}
\]

\[
\begin{align*}
sys &= \text{linearize('aero_guidance_airframe',op,io);} \\
\text{Operating point search report:} \\
\text{Operating point search report for the Model aero_guidance_airframe.} \\
\text{(Time-Varying Components Evaluated at time t=0)} \\
\text{Operating point specifications were successfully met.} \\
\text{States:} \\
\text{(1.) aero_guidance_airframe/Aerodynamics & Equations of Motion/ Equations of Motion (Body Axes)} \\
\begin{align*}
\text{x:} &\quad 0 \quad \text{dx:} \quad 968 \\
\text{x:} &\quad -3.05e+03 \quad \text{dx:} \quad -171
\end{align*}
\text{(2.) aero_guidance_airframe/Aerodynamics & Equations of Motion/ Equations of Motion (Body Axes)} \\
\begin{align*}
\text{x:} &\quad 0 \quad \text{dx:} \quad -0.216
\end{align*}
\text{(3.) aero_guidance_airframe/Aerodynamics & Equations of Motion/ Equations of Motion (Body Axes)} \\
\begin{align*}
\text{x:} &\quad 0 \quad \text{dx:} \quad 968 \\
\text{x:} &\quad -3.05e+03 \quad \text{dx:} \quad -171
\end{align*}
\]
\]
Select Trimmed States, Create LTI Object, and Plot Bode Response

```matlab
set(airframe,'inputname',{'Elevator'},...
    'outputname',{{'az'} {'q'}});
ltiview('bode',airframe);
bdclose('aero_guidance_airframe');
```
**Bode Diagram**

*From: Elevator*

![Bode Diagram](image)

Linear System Analyzer
Anti-Windup Control Using a PID Controller

Model Description

This example shows how to use anti-windup schemes to prevent integration wind-up in PID controllers when the actuators are saturated. We use the PID Controller block in Simulink® which features two built-in anti-windup methods, back-calculation and clamping, as well as a tracking mode to handle more complex scenarios.

The plant to be controlled is a saturated first-order process with dead-time.

We start by opening the model.

Figure 1: Simulink model of PID control of a plant with input saturation. To open this model, type sldemo_antiwindup in a MATLAB® terminal.

The PID Controller has been tuned with saturation ignored using the PID tuner of Simulink® Control Design™.

The controlled plant is a first-order process with dead-time described by

\[ P(s) = \frac{1}{10s + 1} e^{-2s} \]
The plant has known input saturation limits of $[-10, 10]$, which are accounted for in the Saturation block labeled Plant Actuator. The PID Controller block in Simulink features two built-in anti-windup methods that allow the PID Controller block to account for the available information about the plant input saturation.

**Performance Without Using Anti-Windup**

First, we examine the effect of saturation on the closed-loop when the saturation model is not considered by the PID Controller block. Simulating the model in Figure 1 generates the results shown below.

![Figure 2: Setpoint vs. measured output with no anti-windup.](image)
Figure 3: Controller output and saturated input with no anti-windup.

Figures 2 and 3 highlight two of the problems that arise when controlling a system with input saturation:

1. When the setpoint value is 10, the PID control signal reaches a steady-state at about 24, outside the range of the actuator. The controller is therefore operating in a nonlinear region where increasing the control signal has no effect on the system output, a condition known as winding up. Note that the dc-gain of the plant is unity, and therefore there is no reason for the controller output to have a steady-state value outside the actuator's range.
When the setpoint value becomes 5, there is a considerable delay before the PID controller output returns to within the actuator range.

Designing the PID controller to account for the effect of saturation will improve its performance by allowing it to operate in the linear region most of the time and recover quickly from nonlinearity. Anti-windup circuitry is one way to achieve this.

**Configuring the Block for Anti-Windup Based on Back-Calculation**

The back-calculation anti-windup method uses a feedback loop to discharge the PID Controller's internal integrator when the controller hits specified saturation limits and enters nonlinear operation. To enable anti-windup, go to the **PID Advanced** tab in the block's dialog; select **Limit output**; and enter the plant's saturation limits. Then, select **back-calculation** from the **Anti-windup method** menu and specify the back-calculation gain $K_b$. The inverse of this gain is the time constant of the anti-windup loop. In this example, the back-calculation gain is chosen to be 1. For more information on how to choose this value, see Reference [1].

![Image of PID Advanced settings](image)

**Figure 4:** Enabling the back-calculation anti-windup method.

Once back-calculation is enabled, the block has an internal tracking loop that discharges the Integrator output.
**Figure 5:** Under-mask view of the PID Controller block with back_calculation.

Figures 6 and 7 illustrate the result of simulating the model with anti-windup activated. Note how quickly the PID control signal returns to the linear region and how fast the loop recovers from saturation.
Anti-Windup PID Control Demonstration

Copyright 2009-2012 The MathWorks, Inc.
Figure 6: Setpoint vs. measured output with back-calculation.
Figure 7: Controller output and saturated input with back-calculation.

Figure 7 shows that the controller output $u(t)$ and the saturated input $\text{SAT}(u)$ coincide with each other because Limit output is enabled.

To better visualize the effect of anti-windup, Figure 8 illustrates the plant measured output $y(t)$ with and without anti-windup.
Figure 8: Measured output with and without anti-windup.

Configuring the Block for Anti-Windup Based on Integrator Clamping

Another commonly used anti-windup strategy is based on conditional integration. To enable anti-windup, go to the PID Advanced tab in the block's dialog; select Limit output; and enter the plant's saturation limits. Then, select clamping from the Anti-windup method menu.
Figure 9: Setpoint vs. measured output with clamping.
Figure 10: Controller output and saturated input with clamping.

Figure 10 shows that the controller output \( u(t) \) and the saturated input \( \text{SAT}(u) \) coincide with each other because \textbf{Limit output} is enabled.

For more information on when to use \textit{clamping}, see Reference [1].

**Using Tracking Mode to Handle Complex Anti-Windup Scenarios**

The previously discussed anti-windup strategies relied on built-in methods to process the saturation information provided to the block via its dialog. For those built-in techniques to work as intended, two conditions must be met:
The plant’s saturation limits are known and can be entered into the dialog of the block.

The PID Controller output signal is the only signal feeding the actuator.

These conditions may be restrictive when handling general anti-windup scenarios. The PID Controller block features a tracking mode that allows the user to set up a back-calculation anti-windup loop externally. The following two examples are considered to illustrate the use of tracking mode for anti-windup purposes:

1. Anti-windup for saturated actuators with cascaded dynamics
2. Anti-windup for PID control with Feedforward

**Constructing Anti-Windup Circuitry for Saturated Actuators with Cascaded Dynamics**

In the following model, the actuator has complex dynamics. This is common when an actuator has its own closed-loop dynamics. The PID controller is in an outer loop and sees the actuator dynamics as an inner loop, or simply a cascaded saturated dynamics as shown in Figure 1.

**Figure 11:** Simulink model of PID controller with cascaded actuator dynamics.

To open this model, type `sldemo_antiwindupactuator` in a MATLAB terminal.
In this case, a successful anti-windup strategy requires feeding back the actuator output to the tracking port of the PID Controller block as shown in Figure 11. To configure the tracking mode of the PID Controller block, go to the PID Advanced tab in the block's dialog; select Enable tracking mode; and specify the gain $K_t$. The inverse of this gain is the time constant of the tracking loop. For more information on how to choose this gain, see Reference [1].

![Figure 12: Enabling the tracking mode of the PID Controller block.](image)

Figures 13 and 14 show that the plant's measured output $y(t)$ and the controller output $u(t)$ are responding almost immediately to changes in the setpoint. Without the anti-windup circuit, these responses would be sluggish with long delays.
Figure 13: Setpoint vs. measured output.
Figure 14: Controller output and effective saturated input.

Constructing Anti-Windup Circuitry for PID Control with Feedforward

In another common control configuration, the actuator receives a control signal that is a combination of a PID control signal and a feedforward control signal.

To accurately build a back-calculation anti-windup loop, the tracking signal should subtract the contribution of the feedforward signal. This allows the PID Controller block to know its share of the effective control signal applied to the actuator.

The following model includes a feedforward control.
Figure 15: Simulink model of PID controller with feedforward, and plant input saturation.

The feedforward gain is selected to be unity here because the plant has a dc-gain of 1.

To open this model, type `sldemo_antiwindupfeedforward` in a MATLAB terminal.

Figures 16 and 17 show that the plant's measured output $y(t)$ and the controller output $u(t)$ are responding almost immediately to changes in the setpoint. When the setpoint value is 10, note how in Figure 17 the controller output $u(t)$ reduces to be within the range of the actuator.
Figure 16: Setpoint vs. measured output without anti-windup.
Summary

The PID Controller block supports several features that allow it to handle controller windup issues under commonly encountered industrial scenarios.

References

Bumpless Control Transfer Between Manual and PID Control

Model Description

This example shows how to achieve bumpless control transfer when switching from manual control to PID control. We use the PID Controller block in Simulink® to control a first-order process with dead-time.

We start by opening the model.

Figure 1: Simulink model of PID control with bumpless transfer.

To open this model, type sldemo_bumpless in a MATLAB® terminal.

The PID Controller has been tuned with saturation ignored using the PID tuner of Simulink® Control Design™.

The controlled plant is a first-order process with dead-time described by
\[ P(s) = \frac{4}{2.5s + 1}e^{-s} \]

For several operational reasons, the engineers decided to start the control process in an open-loop manner by feeding the plant input with a saturating ramp signal to drive the output of the plant slowly to a desired steady-state value of 40. A control transfer is scheduled to happen at \( t = 150 \). This transition between open-loop control and closed-loop control therefore involves two control phases of operation:

1. **Manual**: A saturated ramp signal feeds the plant input during start-up until \( t = 150 \).
2. **Automatic**: A PID controller will engage the plant at \( t = 150 \), and must take over the process without introducing bumps at the plant input.

To support smooth control transition, the PID Controller block supports two modes of operation: a **tracking mode** and a **control mode**. In control mode, the PID Controller block operates as an ordinary PID controller. In tracking mode, however, the block has an extra input that allows the PID block to adjust its internal state by changing its integrator output so that the block output tracks a prescribed signal feeding this extra input port.

To achieve bumpless control transfer, the PID Controller block must be in tracking mode when the plant is in the manual control phase (open-loop control), and in control mode when the plant is in the automatic control phase (closed-loop control).

**Configuring the Block for Tracking Mode**

To activate signal tracking, go to the **PID Advanced** tab in the block's dialog; select **Enable tracking mode**, and specify the gain \( K_t \). The inverse of this gain is the time constant of the tracking loop. For more information on how to choose this gain, see Reference [1].
**Figure 2:** Enabling the tracking mode of the PID Controller block.

As shown in Figure 1, once tracking mode is enabled, the block has a second input port denoted by TR. Internally this new port is wired as shown under mask:
**Figure 3:** Under-mask view of the PID Controller block with tracking mode.

**Setting Up the Switching Mechanism**

In addition to enabling tracking mode for the PID Controller block, a switching mechanism is needed to achieve the control transfer. Switch 1 determines which signal feeds the plant input and feeds the tracking port of the PID Controller block.

At time $t = 0$, Switch 1 directs the manual control signal to the plant input and the tracking port of the PID Controller block. This allows the output of the PID Controller block to track the manual control signal during the manual phase by adjusting the PID Controller's internal integrator. When control transfer occurs, therefore, the PID Controller output will be approximately the same as the manual control signal.

At time $t = 150$, Switch 1 switches, directing the output of the PID Controller block to the plant input and the PID Controller block's tracking input. The PID Controller block now tracks its own output, which is equivalent to control mode.
Simulating the Bumpless Control Transfer

The setpoint signal and the closed-loop response of the model are shown in Figure 4.

Bumpless Control Transfer Demonstration

Copyright 2009-2018 The MathWorks, Inc.
Figure 4: Setpoint vs. measured output.

Figure 4 clearly shows that the measured output tracks the Setpoint profile without any output bumps at the time of switching (t = 150). To investigate this further, the plant input, control signals are shown in Figure 5.
Figure 5: Control signal switching.
Figures 5 and 6 show that at the switching instance, the plant input has experienced no step changes (bumps), and therefore the control transfer happens in smooth bumpless fashion as intended.

To see the significance of the bumpless transfer setup, consider the case where tracking mode is not used. In this case, the following setup is obtained:

**Figure 6:** Plant input.
Figure 7: Simulink model of PID control with no bumpless transfer.

To open this model, type `sldemo_bumplessno` in a MATLAB terminal.

Figures 8 and 9 show the performance in the absence of an appropriate bumpless control transfer strategy.
Figure 8: Setpoint vs. measured output.
Figure 9: Control signal switching.

It is evident from Figures 8 and 9 that allowing the PID controller to float while the plant is under manual control can result in undesirable large transients upon switching.

**Summary**

As this example shows, the PID Controller block supports bumpless control transfer through the use of tracking mode.
References

Two Degree-of-Freedom PID Control for Setpoint Tracking

Model Description

This example shows how to regulate the speed of an electric motor using two degrees-of-freedom PID control with set-point weighting. We use the PID Controller (2DOF) block in Simulink® as shown below.
Figure 1: Simulink model with two degree-of-freedom PID control of a DC motor.

To open this model, type `sldemo_pid2dof` in a MATLAB® terminal.

The electric motor is an armature-controlled DC motor. Voltage input controls the shaft speed of the motor. The block diagram of the motor is shown in Figure 2. The motor experiences a load torque $T_d$ (0-5 Nm).
**Two Degree-of-Freedom PID Control**

In contrast to the PID Controller block, the PID Controller (2DOF) block provides an extra degree of freedom to allow users to weight the setpoint as it passes through the proportional action channel and the derivative action channel. See the PID Controller (2DOF) help page or type `doc('PID Controller 2DOF')` in a MATLAB terminal for more details. The schematic of the PID Controller (2DOF) appearing in the model is shown below.

**Figure 2:** Block diagram of the motor.
As shown in Figure 3, the error signal seen by the proportional action is given by

\[ b \ast r - y \]

The signal seen by the derivative action is

\[ c \ast r - y \]

and the signal seen by the integral action is

\[ r - y \]

In general, the setpoint weight \( c \) is chosen to be 0 to prevent undesirable transients upon a change in the setpoint, which is an effect known as derivative kick. The setpoint \( b \) affects the overshoot performance of the controller. Generally, a small \( b \) value reduces overshoot. However, smaller \( b \) values can also result in slower response to setpoint changes. For more details on picking the right setpoint values, see Reference [1].
When $b = 1$ and $c = 1$, the behavior of the two degree-of-freedom PID controller is identical to a classical PID controller.

**Simulating with $b = 1$ and $c = 1$**

When $b = 1$ and $c = 1$, the behavior of the two degree-of-freedom PID controller is identical to a classical PID controller. The control signal, setpoint signal, and closed-loop response of the model are shown in Figure 4.
Figure 4: Control signal, setpoint vs. measured output.

Figure 4 clearly shows the spikes in the control signal, which are caused by aggressive proportional and derivative response to the setpoint change. Modifying the $b$ and $c$ weights can make this response less aggressive, as shown next.
**Simulating with the $b = 0$ and $c = 0$**

In this case, the two degree-of-freedom PID controller is known as I-PD where only the I action acts on the classical error signal, and the PD action acts only on the measured output.

![Control signal, setpoint vs. measured output.](image)

**Figure 5:** Control signal, setpoint vs. measured output.

The simulation results clearly show the absence of large transients in the control signal due to the sudden changes in the setpoint.
See Reference [1] for more information on how $b$ and $c$ are chosen.

**Summary**

The PID Controller (2DOF) block in Simulink supports two degree-of-freedom PID control. This block can be used for tracking complex setpoint profiles and moderating the impact of sudden setpoint changes on control signal transients. The PID tuner of Simulink® Control Design™ can be used to automatically tune all the gains ($P, I, D, N, b, c$) of the PID Controller (2DOF) block.

**References**

Data Typing in Simulink

This example shows how to use data types in Simulink. The model used in this example converts a double-precision sine wave having an amplitude of 150 to various data types and displays the converted signals on two scopes.

The first scope shows the original sine wave and the result of converting this signal to signed and unsigned 8-bit integer data types with saturation enabled. The bottom axes show the boolean results of comparing the signed and unsigned integer values.

The second scope shows the original sine wave and the result of converting this signal to two fixed-point data types: one high-resolution, the other lower resolution. The bottom axes show the result of mapping the sine wave to an enumerated data type that indicates whether the signal is positive, negative or zero.

Similar data type support exists for parameters and states. Simulink also supports other integer and floating-point data types, such as, single, int16, and uint32, and fixed-point types with slope/bias and wordlengths from 1 to 128 bits. You can also define types for bus signals and aliases for existing types.

NOTE: If you do not have a license for Fixed-Point Designer™ then the fixed-point signals will be replaced by single-precision signals.
Data Typing in Simulink
Data Typing Filter

This example shows how a filter can be applied to signals of different data types using standard Simulink® blocks. In this example, a signal with some noise on it is produced. This signal is then fed into a filter as a single precision data type and a standard double data type. The output from the filter for each signal is then compared and the error is displayed on the scope.
Simulink Featured Examples

![Scope window](image)

- **Reduced Precision**
- **Error**
- **Full Precision**

Graph showing waveforms under different precision conditions.
Simulink Bus Signals

This example introduces you to the following Simulink® bus signal capabilities:

- Bus Creator, Bus Selector, and Bus Assignment blocks
- Bus Element Ports at component interfaces
- Smart editing to perform common bus workflows faster

The three areas in the slexBusExample model used in this example correspond to these three capabilities.

Bus Creator, Bus Selector, and Bus Assignment Blocks

The subsystem in the "Bus Creator, Bus Selector, and Bus Assignment Blocks" area shows how to

Tips & Tricks For Working With Buses

"Bus Element Ports" action
"Create Bus" action
Automatic Port Creation


15-385
• Create bus signals using the Bus Creator block
• Replace signals in a bus using the Bus Assignment block
• Select signals from a bus using the Bus Selector block

Use a Bus Creator block to bundle the signals connected to its inputs into a bus. A bus signal represents a set of signals, analogous to a bundle of wires tied together. For example, the bus signal created by Bus Creator 1 contains the two signals connected to its inputs: sine and chirp. You can view the hierarchy of a signal by right-clicking on it and selecting Signal Hierarchy from the menu. You can also create nested buses. For example, sinusoidal and non-sinusoidal are sub-bus signals in the bus created by Bus Creator 3.

Use a Bus Assignment block to replace one or more signals in the signal that is connected to its Bus input. For example, Bus Assignment replaces the signals constant and non-sinusoidal in the bus created by Bus Creator 3 with new signals. You can use the Bus Assignment block to replace both leaf and sub-bus signals.
Use a Bus Selector block to extract one or more signals from the bus connected to its input. For example, Bus Selector selects the non-sinusoidal.pulse, sinusoidal.sine, and constant signals. You can run the simulation to visualize non-sinusoidal.pulse and sinusoidal.sine in the scope and constant in the display.

After running the simulation, look at the lines in slexBusExample. Notice that several signals are drawn with three lines. This line format indicates that the signal is a bus signal.

For more information on buses, see Getting Started with Buses.

**Bus Element Ports at Component Interfaces**

The subsystems in the "Buses at Component Interfaces (Bus Element Ports)" area show how to

- Create bus signals using the Out Bus Element block
- Select signals from a bus using the In Bus Element block

**Out Bus Element Blocks**

The first subsystem consists of five source blocks and five Out Bus Element blocks.
An Out Bus Element block is similar to a Bus Creator block connected to an Outport block. Each Out Bus Element block has a label. You can display this label in two modes. In **Expanded Notation** mode, the label is made up of the name of the corresponding subsystem port and a dot-separated string that determines the element in the bus. For example, the Out Bus Element block with the label `Out1.sinusoidal.sine` creates a leaf-signal named `sine` in a sub-bus named `sinusoidal` at the output port named `Out1`. In **Compact Notation** mode the leaf-signal name becomes the label. You can switch between **Expanded Notation** and **Compact Notation**.

In either mode, you can directly edit the label text to change the element created in the output bus.
To create a new element in the bus, copy and paste an Out Bus Element block. To create a new output port at the subsystem interface, right-click and drag an Out Bus Element block and select **Create New Port**.

To see the bus created by a group of Out Bus Element blocks, double click on the icon of one of the blocks to open the port properties dialog box. You can highlight the signals connected to the blocks, change the name and number of the port, change the colors of the blocks, and reorder the signals in the bus.

**In Bus Element Blocks**

15-389
The second subsystem consists of two Scope blocks, one Display block, and five In Bus Element blocks.

An In Bus Element block is similar to an Inport block connected to a Bus Selector block. The labels of In Bus Element blocks work the same way as Out Bus Element block labels do. For example, the In Bus Element block with the label In1.sinusoidal.sine selects a leaf signal named sine in a sub-bus named sinusoidal.

To change the element selected from the input bus, edit the label text directly. If a bus is already connected to the corresponding input port, you can select from a list of available signals.
To select a new element from the bus, copy and paste an In Bus Element block. To create a new input port at the subsystem interface, right-click and drag an In Bus Element block and select **Create New Port**. To see the bus accessed by a group of In Bus Element blocks, double-click the icon of one of the blocks to open the port properties dialog box. You can change the name and number of the port, change the color of the blocks, and observe any missing or unused signals in the incoming bus.

For more information about using In Bus Element and Out Bus Element blocks, see **Simplify Subsystem Bus Interfaces**.

**Smart Editing to Perform Common Bus Workflows Faster**

The subsystems in the "Tips & Tricks For Working With Buses" area show how to speed up common bus tasks by using:

- The "Bus Element Ports" smart editing cue to convert Bus Selector and Bus Creator blocks to In Bus Element and Out Bus Element blocks
- The "Create Bus" smart editing cue to create buses at subsystem interfaces and to bundle outputs of individual blocks into a bus
- Automatic port creation to add a new signal to a Bus Creator block and to select a new signal from a Bus Selector block

**Convert Bus Creator and Bus Selector blocks to In Bus Element and Out Bus Element Blocks**

You can use the "Bus Element Ports" smart editing cue to convert existing subsystems that use Bus Selector and Bus Creator blocks at their interfaces to use instead In Bus Element and Out Bus Element blocks.

1. Click on a Bus Selector block that is connected to an Inport block or a Bus Creator block that is connected to an Outport block.
2. Select **Bus Element Ports** from the action bar.
**Bundle Individual Signals into a Bus**

You can use the "Create Bus" smart editing cue to bundle multiple signals into a bus.

1. Drag a selection box around the signals of interest.
2. Select **Create Bus** from the action bar.

**Between Subsystems**

When performed between two subsystems, this action replaces the Inport and Outport blocks in the source and destination subsystems with In Bus Element and Out Bus Element blocks.
**Subsystem Inputs**

When performed at the input of a subsystem that is connected to some other non-subsystem block, this action bundles the selected signals into a bus using a Bus Creator block and replaces the Inport blocks in the subsystem with In Bus Element blocks.
**Subsystem Outputs**

When performed at the output of a subsystem that is connected to some other non-subsystem block, this action replaces the Outport blocks in the subsystem with Out Bus Element blocks and inserts a Bus Selector block to maintain connectivity.
**Block Outputs**

When performed at the outputs of individual blocks, this action inserts a Bus Creator block, resizes it, and makes the connections.
Automatic Port Creation

You can add a new signal to a bus by dragging a signal line to a Bus Creator block. To select a new signal from a bus, drag a signal line to a Bus Selector block and choose the signal you want from the list of available signals.
Modeling Arrays of Buses

This example illustrates the use of arrays of buses in Simulink® to represent structured data compactly, eliminating the need to include multiple copies of the same buses. You can also use For Each subsystems to iteratively process each element of the bus array.

In this example, the array of buses is created using a Vector Concatenate block. The input bus signals to the Concatenate block must be nonvirtual and of the same type (that is, have the same names, hierarchies and attributes for the leaf elements).

The example also illustrates the use of the Selector and Assignment blocks to select from and assign into the array of buses.
Matrix Signals

This example shows how to use matrix and frame signals in Simulink® diagrams. Through this example, you can access models that relate to the following practical applications.

- Feedback Systems
- Lunar Module Digital Autopilot
- Matrix Support S-Function
- Frame Signals
Variable-Size Signal Basic Operations

This model shows how variable-size signals can be generated and also illustrates some of the operations that can be applied to them. The purpose of this example is to introduce you to the basic operations associated with variable-size signals.

About This Example

Open the example model sldemo_varsize_basic. This model contains several blocks that support variable-size signals.

Common ways of generating variable-size signals are:

- Use a Switch block with a different size signal at each input port.
- Use a Selector block and allow the size selection index to change over time.

Eighty blocks can operate with variable-size signals including the Gain block, the Sum block, the Math Function block, the Matrix Concatenate block, and the Bus Creator block. In addition, you can probe variable-size signals using the Width or the Probe block, display variable-size signals on Scope blocks and save variable-size signals to the workspace using the To Workspace block.

For a complete list of blocks that support variable-size signals, see "Simulink® Block Support for Variable-Size Signals" in the Simulink documentation.
Variable-Size Signal Length Adaptation

This model shows a hypothetical system where the length of a signal changes over time by adapting to the changes of a control signal.

About This Example

Open the example model `sldemo_varsize_dataLengthAdapt`.

This model consists of two sections: the left section generates a signal and converts the signal to variable size; the right section processes the variable-size signal and outputs it to a scope. Length adaptation is based on the value of the control signal. When the control signal falls within one of the three predefined ranges, the size of the data signal changes accordingly.

This variable-size signal is then fed into a processing block, where blocks supportive of variable-size signals, operate on it. Note the application of a MATLAB Function block with both the input and the output signals of variable-size. The resulting signal, as well as the signal width, are fed to a scope for visualization.
Multimode Variable-Size Signal

This model shows how to use different operation modes to correspond to different signal sizes.

About This Example

Open the example model sldemo_varsize_multimode. The processing subsystem in this model receives a variable-size signal where the size of the signal depends on the operation mode of the system. For each mode change, the Mode Control Logic Stateflow® Chart generates a function-call to reset the blocks contained in the Process subsystem.

Mode-Dependent Variable-Size Signals

This model represents a system that has three operation modes. For each mode, the signal has a different size. One Function-Call subsystem with variable-size signals handles the processing of the different signal sizes for each mode. The Mode Control Logic block detects the data signal length changes, and responds by issuing mode-switching events. This event triggers the reset of the state and the repropagation of signal sizes within the subsystem.
Parallel Channel Power Allocation

This model shows a potential use of the Find block. This block outputs a variable-size signal containing the indices of the nonzero values of the input.

About This Example

Open the example model **sldemo_varsize_channel_pow_alloc**. This model implements the waterfilling algorithm for obtaining the optimal power allocation for communication channels when a power budget exists. The communication channels involved are independent (i.e., parallel channels) and have different noise levels.

The algorithm iteratively increases the value of the waterline, so that the channels which have a noise level less than the waterline are broadcast. The power used by these transmitting channels is equal to the difference between the waterline and the noise level of the channel. The sum of the power used by all the channels transmitting should satisfy the power budget within the specified tolerance.

The Find block is used in the calculation of the total power used for a certain waterline value. The channels which are set to transmit at that waterline level are selected using logical indexing. The logical indices are obtained after thresholding the difference of the waterline and the channel noise levels. As a result, the channels with noise levels larger than the waterline are assigned a zero, and those with noise levels below the waterline are assigned ones. The Find block is used for converting the logical indices into linear indices. Note that this signal is properly modeled by a variable-size signal since the number of channels below the threshold is unknown at compile time. The linear indices are fed into a Selector block that picks the channel transmission powers. Subsequently, these powers are summed to get the current value of used power. If this value is smaller than the power budget, the waterline is increased and the process is repeated until the used power equals the power constraint.
Optimal Power Allocation for Parallel Channels (Waterfilling)

The NL signal contains the channel noise levels. The Pcon signal is the total available power budget. Tol is the tolerance to use in the search for the waterline level.
Merging Signals

This model shows the following Simulink® concepts:

**Model Description**

The left side of this model contains two blocks, for example the Repeating Sequence block, from the Simulink Sources library. Each block generates a scalar output signal of the form displayed on the block's icon.
These two signals are fed into different Enabled Subsystems, which are outlined with thick lines.
The enabled (or conditionally executed) subsystems are controlled by the signal generated by a Discrete Pulse Generator block.
This block's output toggles between the values of zero and one every five seconds. When the output of the Discrete Pulse Generator block is positive, the top enabled subsystem is simulating. When the output of the Discrete Pulse Generator block is zero, the bottom enabled subsystem is simulating.

To force the bottom enabled subsystem to simulate when the Discrete Pulse Generator block's output is zero, the output is passed through a Logical Operator block set to the NOT operator. The NOT block outputs a value of one when its input is zero.
The outputs of the two enabled subsystems are fed into a Merge block.
The output of the Merge block is, finally, displayed on a Simulink Scope.
You can run the simulation to see the output of the Merge block on the Scope. Notice that the type of waveform shown on the Scope changes every five seconds. This is due to the behavior of the Merge block.
Conditionally Executed Subsystems

The Merge block example contains two conditionally executed subsystems.
A conditionally executed subsystem is a subsystem that is only executed at particular times during the simulation, i.e., when certain conditions are met. In this case, the subsystems are enabled subsystems, distinguished by their icons and the additional input port on the top or bottom of the block, as shown below.
The signal feeding into the additional port is called the control signal. The value of this signal provides the condition that determines whether the subsystem is executed, or not, at any given time step. Enabled subsystems are executed when the control signal is strictly positive.

In this example, you want to execute one of the enabled subsystems when the control signal is positive, and the other when the control signal is zero. You can do this by using the Logical Operator block in Simulink.

**Logical Operator Block**

The Logical Operator block in Simulink gives you a method for incorporating logical operators and boolean signals into your Simulink diagram.
You can open the Logical Operator block to see the different operators that are listed in the drop down menu. The current setting of NOT returns 1 (or TRUE) when the input signal is non-zero, or 0 (FALSE) when the input signal is strictly zero.

In this example, you can use this feature to generate a positive value when the control signal goes to zero by maintaining the current operator setting of NOT and closing the dialog.

**Note:** In the Simulink Editor, on the **Debug** tab, the **Information Overlays** menu offers options for displaying signal and port properties on the block diagram.
Boolean Signals in Simulink®

You have the option to have Simulink to use boolean or double values for the inputs and outputs of the Logical Operator block, using the following steps.

1. To open the Configuration Parameters dialog box, in the Simulink Editor, on the Modeling tab, select Model Settings.

2. Go to the Optimization page in the Configuration Parameters dialog.

3. See that the Implement logic signals as boolean data (vs. double) parameter has been enabled. This optimization tells Simulink whether or not it should allow a signal whose data type is double to be passed into and out of blocks that support the boolean data type.

4. Click the OK button on the Configuration Parameters dialog.
By having the Data Type Conversion block after the Discrete Pulse Generator block, you avoid an error indicating that the Logical Operator block expects a boolean input signal.

The Merge Block and Alternately Enabling Subsystems

You can use the Merge block to create a single signal whose value is equal to the output of whichever enabled subsystem is currently executing. The following steps.
1. Open the Merge block's Block Parameter dialog by double clicking on the block.

2. Specify the **Number of inputs** as the number of enabled subsystems whose outputs you want to merge. In this case you want to set the *Number of inputs* to be 2.

3. Leave the **Initial output field** empty to specify that the Merge block sets its initial output to the initial value of one of its input signals. In this case, the output will be the initial value of the subsystem that is enabled when the simulation starts.

4. Close the Merge block's Block Parameter dialog.

5. Connect the output of the two conditionally executed subsystems to the inputs of the Merge block.
As before, when you run the simulation, the output of the Merge block is the value of whatever conditionally executed subsystem is executing at any point in time.
Interesting Feature: Programmatically Changing a Block's Background Color

When you ran the simulation, you should have noticed that the color of the currently executing conditionally executing subsystems changed. This does not happen automatically for all conditionally executing subsystems. It was achieved using an S-function and MATLAB® code!

If you open one of the enabled subsystems by double clicking on the block in the Simulink diagram, you’ll see it contains the following.

1. An Enable block, which is what makes this a conditionally executed subsystem and adds the input port for the control signal
2. A Level-2 MATLAB-file S-function block, which runs the MATLAB file S-function `mergefcn.m`
At each time step, the S-function's mdlUpdate subfunction executes. It sets the BackgroundColor property of the Subsystems based on whether it is currently executing, or not, with the following command.

```matlab
set_param('mergedemo/Subsystem','BackgroundColor','green')
```

For more information on how to write MATLAB file S-functions, see the documentation.
Migration to Structure Parameters

This example shows how to convert a Simulink® model that is parameterized by unstructured workspace variables to a model that is parameterized by a MATLAB® structure. The example uses a number of Simulink utilities to organize model variables from the base workspace into a hierarchical structure and then apply the structure to the model.

**Model Description**

Open the model `sldemo_applyVarStruct`.

This model uses a number of numeric parameters from the base workspace.
Example Requirements

This example changes the original model and creates files in the current working directory. If you plan to save the example models, save a copy in the current directory.

Typical Workflow

Migrating a model to use structure parameters follows these steps:

1. Collect all numeric variables used by the model into a MATLAB structure.
2. Modify the MATLAB structure (if necessary).
3. Apply the MATLAB structure to the model.
4. Configure the appearance of the parameter structure in the generated code (optional).

Collecting Numeric Variables into a MATLAB Structure

You can use Simulink.findVars to find out what variables are used by the Simulink model.

```matlab
model = 'sldemo_applyVarStruct';
varList = Simulink.findVars(model, 'WorkspaceType', 'base')
```

```matlab
varList =
20x1 VariableUsage array with properties:
   Name
   Source
   SourceType
   Users

var = varList(1)

var =
VariableUsage with properties:
   Name: 'Ka'
   Source: 'base workspace'
```
You can use `Simulink.BlockDiagram.createVarStruct` to generate a MATLAB structure containing all numeric variables from the base workspace or data dictionary used by the model.

```matlab
Simulink.BlockDiagram.createVarStruct(model, 'ModelParam')
```

Created variable 'ModelParam' in base workspace.

Variable `ModelParam` is a MATLAB structure that contains the numeric variables as fields (sorted by name).

**NOTE:**

- The model and all data that the model uses must be loaded before calling `Simulink.BlockDiagram.createVarStruct`.
- You cannot give the MATLAB structure the same name as an existing variable.

### Modifying the MATLAB Structure

The original structure contains all the variables in one level of hierarchy. You can change the structure before applying it to the model. For example, reorder parameter fields, add extra levels to group model parameters, or add extra fields.

**NOTE:**

- Do not change the names of the leaf fields in the structure.
- Do not clear the original variables until you have finished the migration process.

1. One way to do this is to use `Simulink.saveVars` to write the structure to a MATLAB script and make changes in the file.

```matlab
Simulink.saveVars('sldemo_applyVarStruct_data.m','ModelParam')
edit sldemo_applyVarStruct_data.m
```

2. Edit the MATLAB script as needed. For example, split the structure into two structures.

- `'ControlParam'` contains all parameters for the Controller subsystem.
- `'ModelParam'` contains all other numeric parameters.
ControlParam = struct;
ControlParam.Ka = 0.677;
ControlParam.Kf = -1.746;
ControlParam.Ki = -3.864;
ControlParam.Kq = 0.8156;
ControlParam.Tal = [0.3959 1];
ControlParam.Ts = [0.1 1];
ControlParam.W1 = [1 2.971];
ControlParam.W2 = [1 4.144];

ModelParam = struct;
ModelParam.Md = -6.8847;
ModelParam.Mq = -0.6571;
ModelParam.Mw = -0.00592;
ModelParam.Swg = 3;
ModelParam.Ta = 0.05;
ModelParam.Uo = 689.4;
ModelParam.Vto = 690.4;
ModelParam.Zd = -63.9979;
ModelParam.Zw = -0.6385;
ModelParam.a = 2.5348;
ModelParam.b = 64.13;
ModelParam.g = 32.2;

Save the modified data file. You can use it as a preload function.

3. Run the modified MATLAB script to create the MATLAB structures.
sldemo_applyVarStruct_data;

Applying the MATLAB Structure to the Model

You can use Simulink.BlockDiagram.applyVarStruct to automatically replace the variable references in the block diagram by the equivalent structure references.

1. Apply the MATLAB structure to the model. The model will be changed. You can then save it to a writable directory.

    [applied,unapplied]=Simulink.BlockDiagram.applyVarStruct(model,'ControlParam')
    [applied,unapplied]=Simulink.BlockDiagram.applyVarStruct(model,'ModelParam')

**NOTE:**

- You can use 'preview' ApplyMode to get information about what will be changed without actually changing the model:
Simulink.BlockDiagram.applyVarStruct(model, 'ModelParam', 'ApplyMode', 'preview')

- If the model has been compiled, you can use 'cached' SearchMethod to avoid model recompiling:
  
  Simulink.BlockDiagram.applyVarStruct(model, 'ModelParam', 'SearchMethod', 'cached')

- If unapplied output is not empty, you can investigate the reason using Simulink.findVars(model, 'WorkspaceType', 'base', 'Name', variableName). The probable causes are: you changed structure field names or added extra fields; the variable is used by a block that needs special consideration, e.g., Stateflow® chart, S-Function or Model block.

2. Test the MATLAB structure in the modified model.

Clear unnecessary variables and run the simulation to check results.

```matlab
clear;
model = 'sldemo_applyVarStruct';
sldemo_applyVarStruct_data;
sim('sldemo_applyVarStruct');
```

- Check for any warnings or errors.
- Check simulation results.

**Configuring the Structure Parameter to Appear in Generated Code**

By default, the MATLAB structure does not appear in generated code because numeric values are inlined. Code generation requires Simulink® Coder™.

1. To make the structure variable tunable, create Simulink.Parameter with the structure value and 'ExportedGlobal' storage class.

```matlab
p = Simulink.Parameter);
p.Value = ControlParam;
p.CoderInfo.StorageClass = 'ExportedGlobal';
ControlParam = p;
clear p;
rtwbuild('sldemo_applyVarStruct');
```

The generated code now contains a structure type definition for the tunable structure in the model_types.h header file. By default, Simulink Coder generates an automatic name for the type of parameter structures. This name is unique, but not easy to recognize.
typedef struct {
    real_T Ka;
    real_T Kf;
    real_T Ki;
    real_T Kq;
    real_T Tal[2];
    real_T Ts[2];
    real_T W1[2];
    real_T W2[2];
} struct_mn4cJ7zsH8aqc8bBlbIkqC;

2. You can control this type name by using Simulink.Bus to specify the data type of the Simulink.Parameter object.

You can use Simulink.Bus.createObject to create a bus object with the same shape as the MATLAB structure.

busInfo=Simulink.Bus.createObject(ControlParam.Value)

Assign the bus object name to the Simulink.Parameter DataType attribute.

ParamType=eval(busInfo.busName);
ControlParam.DataType='Bus: ParamType';
clear(busInfo.busName);
clear busInfo;

NOTE: Only Simulink.Parameter can accept the bus object name as a data type. For MATLAB structures, Simulink Coder generates an automatic name for the type.

3. Resave the data file to include the modified workspace variables.

Simulink.saveVars('sldemo_applyVarStruct_data.m', '-append')

4. Regenerate code for the modified model (requires Simulink Coder).

rtwbuild(model);

The generated type definition now appears as:

typedef struct {
    real_T Ka;
    real_T Kf;
    real_T Ki;
    real_T Kq;
    real_T Tal[2];
}
real_T Ts[2];
real_T W1[2];
real_T W2[2];
} ParamType;

The parameter definition is:

ParamType ControlParam = {
  0.677,
  -1.746,
  -3.864,
  0.8156,
  {0.3959, 1.0},
  {0.1, 1.0},
  {1.0, 2.971},
  {1.0, 4.144}
}
Share Data Store Between Instances of a Reusable Algorithm

To reuse an algorithm, instead of copying and pasting the blocks, you can encapsulate them in a separate model file. Then, you can refer to the model repeatedly with Model blocks. Each Model block is a standalone instance of the algorithm.

By default, each instance calculates and stores its own copy of the signals that the model contains. Therefore, the instances do not interact. However, you can use a data store to share a piece of data between the instances.

This example shows how to share a fault indication between all instances of a reusable control algorithm. Any instance can set the fault, causing all instances to take action to clear it.

Explore Example Models

Open the example model sldemo_shared_data.

The Controller subsystem refers to the controller algorithm model, sldemo_shared_data_alg, multiple times through three Model blocks.

Open the algorithm model.
The Data Store Memory block `fault_ind` represents a fault indication. The block parameter **Signal Attributes > Share across model instances** is selected, so all instances of `sldemo_shared_data_alg` share the indication. Because the block exists in `sldemo_shared_data_alg`, the indication is local to the model. Other models that refer to `sldemo_shared_data_alg` cannot read or write to the indication.

The Algorithm subsystem uses a PID controller to generate a position command for an actuator (represented by a block in `sldemo_shared_data`).

The Fault Handling subsystem sets and clears the fault indication. Each time an instance of the algorithm experiences a fault condition (the controlled input exceeds 2.63), the subsystem increments the value of the data store by 1. Each time an instance clears its fault condition, the subsystem decrements the value by 1.
Simulate Example Models

Simulate the parent model, sldemo_shared_data.

Use the Simulation Data Inspector to observe the position commands and the controlled signals. In the Simulink Editor toolbar, click the highlighted button.

Partway through the simulation, the controlled signal conc_mid experiences a disturbance.

Current plot held
The disturbance triggers a fault in the corresponding instance of the control algorithm. Due to the fault, all three position commands (the vLV signals) lock at zero.
Later, when the fault condition clears, all of the position commands and controlled signals return to nominal values.
Share Data Store Between Instances of a Reusable Algorithm
Attaching Input Data to External Inputs via Custom Input Mappings

This example shows how to create a custom mapping function for the Root Inport Mapper tool. The Root Inport Mapper tool associates MAT-file data with a specific input port, based on one of these criteria.

- **Port Order** - Maps in the order it appears in the file to the corresponding port number.
- **Block Name** - Maps by variable name to the corresponding root inport with the matching block name.
- **Signal Name** - Maps by variable name to the corresponding root inport with the matching signal name.
- **Block Path** - Maps by the BlockPath parameter to the corresponding root inport with the matching block path.
- **Custom** - Maps using a MATLAB function.

Use a custom mapping criteria when the data you are trying to map does not match any of the other mapping criteria.

**Custom Mapping Scenario**

If the combination of input data and the model input ports does not match Port Order, Block Name, Signal Name, or Block Path mapping criteria, you can do one of the following:

- Rename the input data in the MAT-file
- Rename the input ports on the model
- Use the custom mapping mode

This example shows how to write a custom mapping function.

The workflow described below includes

- Create a MATLAB File to contain the custom mapping function
- Declare the custom mapping function name, inputs, and outputs
- Initialize the output for assignment of the return value
• Define and implement your mapping algorithm
• Validate your mapping algorithm

Create a New MATLAB Function File

Create a MATLAB function file. This file will contain your custom mapping function.

Declare Custom Mapping Function

Declare a custom mapping function name, and specify the inputs and outputs. slexCustomMappingMyCustomMap is the custom mapping function in this example. It declares the function inputs and outputs as defined below.

Inputs

• modelName - String that contains the model name.
• signalNames - Cell array of strings that contain variable names of signals to map
• signals - Cell array of signal data aligned with signalNames such that signalNames{1} is the variable name of the signal stored in signals{1}.

Outputs

• inputMap - Array or scalar of objects that result from the getRootInportMap function.

Initialize the Output

Initialize the value of the output variable inputMap. To do this use the getRootInportMap command with the empty property as shown in the example function slexCustomMappingMyCustomMap or in the code snippet below. This initializes the output variable to empty so that a value can be assigned to the output.

inputMap = getRootInportMap('empty');

Define Your Custom Mapping Algorithm

Things to think about when defining your custom mapping algorithm.

• What is the criteria for mapping signals to a root-level inport, for example BlockName mapping matches the name of the input signal to the name of the root-level inport block name.
• What information is needed to create a mapping.
• What happens if the mapping criteria determines that there is no match between signal and root-level inport.

The example addresses these items by
• Using the root-level port number and signal name for mapping criteria.
• Using a utility function slexCustomMappingGetPortNames to get root-level inport block names to provide as input along with the variable |signalName| to the getRootInportMap function which creates the mapping.
• Including a logical condition to handle the case where there is no match between port number and signal name.

**Implement Your Mapping Algorithm**

Next, implement your mapping algorithm.

In the slexCustomMappingMyCustomMap implementation, the root-level inport port number is the mapping criteria criteria. The algorithm gets this port number from the root-level inport block. It then gets the numerical values appended to the names of the signals that are passed into the function, and compares the port number with the numerical values.

**NOTE:** In addition to implementing your mapping algorithm, you should also include a logical statement to handle situations where a mapping is not possible. The example function does this by using an if statement with the isempty function to determine if a mapping can be made.

**Create the Mapping**

The final step is to use the getRootInportMap function to create a custom map. The getRootInportMap function uses Property-Value pairs to create a custom map. The properties are:

• *model* - String that represents the model name.
• *signalName* - String or cell of strings that contains the value of the variable name of the signal to map.
• *blockName* - String or cell of strings that contains the value of the block name of the port to be mapped.

The following code is an example of this function. This same code appears in the example function slexCustomMappingMyCustomMap.
inputMap = getRootInportMap(
    'model', modelName,
    'signalName', signalNames,
    'blockName', portNames);

Validate the Custom Mapping Function

To validate the custom mapping function at the command line:

1. Open the model against which you want to test your function.
2. Create some input signals to support the model.
3. Call your custom mapping function with the correct input parameters.
4. Validate the expected input string to send to the `sim` command, or to place in the model configuration parameters External Input String edit box.

Execute the lines of code in the following example. This code uses the custom mapping function provided with this example. If the custom map function succeeds, you should see a comma-separated list of variable names displayed in the order of the port number to which they are assigned (`'port1, port2'`).

```matlab
load_system('slexAutotransRootInportsExample');
port1 = timeseries(ones(10,1)*10);
port2 = timeseries(zeros(10,1));
inputMap = slexCustomMappingMyCustomMap('slexAutotransRootInportsExample',
                               {'port1', 'port2'},
                               {port1, port2});
externalInputString = getInputString(inputMap, 'base');
close_system('slexAutotransRootInportsExample',0);
```

If your signals are in a Simulink.SimulationData.Dataset, you can execute the lines of code in the following example. This code uses the custom mapping function provided with this example. If the custom mapping function succeeds, you should see a comma-separated list of variable names displayed in the order of the port number to which they are assigned (`'ds.getElement(1), port1, ds.getElement(2)'`).

```matlab
load_system('slexAutotransRootInportsExample');
ds = Simulink.SimulationData.Dataset;
ds = ds.addElement( timeseries(ones(10,1)*10), 'port1');
ds = ds.addElement( timeseries(zeros(10,1)), 'port1');
inputMap = slexCustomMappingMyCustomMap('slexAutotransRootInportsExample',
                               {'ds'},
                               { ds });
externalInputString = getInputString(inputMap, 'base');
close_system('slexAutotransRootInportsExample',0);
```
Using Mapping Modes with Custom-Mapped External Inputs

This example shows how to implement a custom mapping algorithm similar to a Simulink mapping mode. It uses the `getSlRootInportMap` and `getRootInportMap` functions to implement the custom mapping.

This example assumes that you are familiar with the `getRootInportMap` command and the Root Inport Mapper Tool custom mapping capability. If you are not familiar with those concepts, read the documentation and review the examples that pertain to `getRootInportMap` and custom mappings.

The Workflow

This example shows how you can use a built-in Simulink mapping mode to perform as many mappings as possible. It then flags the root inports that were not able to be assigned a signal. The algorithm then overrides the flagged mappings with custom mappings to map the remaining signals. To implement such a solution, create a custom mapping function using the `getSlRootInportMap`.

This example uses a list of inputs with two kinds of signals:

- Signals that can be mapped using the Simulink block name mapping mode.
- Signals that cannot be mapped using the Simulink block name mapping mode. You must map these signals with a custom mapping mode.

Assume the following scenario:

- You want to use a group of signals as inputs to your Simulink model.
- The signals are named such that the variable names match the block name of the root-level inport.
- Each signal that uses this naming convention is within tolerance.
- Each signal that has the 'x' character appended to its name is considered outside tolerance.

This example uses a mapping mode similar to the Simulink block name mapping method.

The root-level inport block names are:
• Throttle
• Brake

The signal variable names are:
• Throttlex
• Brake

To map inputs to root-level inport blocks in this scenario, you need a custom mapping function for the Root Inport Mapper tool. This example uses the AlmostBlockName custom mapping function.

For this example, you will use the slexAutotransRootInportsExample model to validate your custom mapping function.

**Declare the Custom Mapping Function**

Declare the function name, inputs, and outputs. To do this, copy and paste the following code snippet into a MATLAB file and save the file as AlmostBlockName.m.

```matlab
function inputMap = AlmostBlockName( modelName, signalNames, signals )
```

**Get the Simulink BlockName Mapping**

Next, map all the signals. To do this, first map all the signals within tolerance using the Simulink block name mapping mode, then map the signals outside tolerance.

To map signals within tolerance to a model using one of the Simulink mapping modes, use the function `getSlRootInportMap`. This function returns the `inputMap` and a vector of logical values. Each logical value indicates a successful or unsuccessful mapping of `inputMap` to a signal. To map by block name, insert the following lines of code just after the function declaration.

```matlab
inputMap = getRootInportMap('empty');
if ~bdIsLoaded(modelName)
    load_system(modelName);
end

[inputMap, hasASignal] = getSlRootInportMap('Model', modelName, ...
    'MappingMode','BlockName',...
    'signalName',signalNames, 'signalValue', signals);
```
Find the Missing Input Signals

In the previous step, you created a mapping using a block name mapping mode. You must now account for an empty `inputMap` and for `inputMap(s)` that were not associated with a signal within tolerance. The function `getSlRootInportMap` has flagged these signals with the output variable `hasASignal`. To do this:

1. Check the `inputMap` variable.
2. If the `inputMap` variable is not empty, determine which elements of the `inputMap` vector were not assigned a signal. To do this, use the logical `~` on the `hasASignal` vector as shown below. The `emptyIndex` vector now contains a logical vector where `true` means the `inputMap` does not have a signal mapped to it.
3. Copy and paste the following code snippet under the call to the `getSlRootInportMap` and before the end to the if `bdIsLoaded(modelName)`.

```matlab
if ~isempty(inputMap)
    emptyIndex = ~hasASignal;
end
```

The code snippet performs steps one and two for you.

Complete the Mapping

In the previous step, you created a logical vector `emptyIndex` to see if any of the `inputMap` objects were not associated to a signal. If all the elements of the `emptyIndex` vector are false, you have a complete mapping and the code added in this section will not be executed.

If the `emptyIndex` vector contains at least one value that is true, you have `inputMap` objects that are not associated to a signal. Manually assign the variable signal(s) to that `inputMap`. Then, override the `inputMap` with the signal name that matches the expected signal name:

1. In the `emptyIndex` vector, find all the items that are true. These items point to the `inputMap(s)` that still need to be associated with a signal.
2. For each `inputMap`, use the 'BlockName' property to get the name of the inport block that the `inputMap` is assigned to.
3. Append an 'x' to the block name to get the name of the signal to be assigned to the `inputMap`.
4. Compare the result to each item in the `signalNames` variable cell array.
If a match is found, override the inputMap with the signal name that matches the expected signal name. To override the inputMap object, use the getRootInportMap function with the 'InputMap' and the 'SignalName' properties.

```matlab
if isa( signals{1}, 'Simulink.SimulationData.Dataset')
    signalNames = signals{1}.getElementNames';
end

idxEmpty = find(emptyIndex==true);
for kEmpty =1:length(idxEmpty)
    idxOfEmpty = idxEmpty(kEmpty);
    destBlockName = get(inputMap(idxOfEmpty),'BlockName');
    outSideToleranceSig = [destBlockName 'x'];
    isAMatch = strcmp(signalNames, outSideToleranceSig);
    if any(isAMatch)
        inputMap(idxOfEmpty) = getRootInportMap('InputMap', ... 
            inputMap(idxOfEmpty), 'SignalName', outSideToleranceSig);
    end
end
```

### The Custom Map File

When you are done, the file AlmostBlockName.m should resemble the following code.

```matlab
function inputMap = AlmostBlockName(modelName, signalNames, signals)
    inputMap = getRootInportMap('empty');
    if bdIsLoaded(modelName)
        [inputMap, hasASignal] = getSlRootInportMap('Model', modelName, ... 
            'MappingMode', 'BlockName', ... 
            'signalName', signalNames, 'signalValue', signals);
        if isempty(inputMap)
            emptyIndex = ~hasASignal;
            idxEmpty = find(emptyIndex==1);
            if isa( signals{1}, 'Simulink.SimulationData.Dataset')
                signalNames = signals{1}.getElementNames';
            end
            for kEmpty =1:length(idxEmpty)
                idxOfEmpty = idxEmpty(kEmpty);
                destBlockName = get(inputMap(idxOfEmpty), 'BlockName');
                nonNominalSig = [destBlockName 'x'];
                isAMatch = strcmp(signalNames, nonNominalSig);
                if any(isAMatch)
```

15-445
inputMap(idxOfEmpty) = getRootInportMap('InputMap', ... 
   inputMap(idxOfEmpty),'SignalName',nonNominalSig);
end
end
end

**Validate the Custom Mapping**

To validate your custom mapping:

1. Save the AlmostBlockName function in a file on the MATLAB path.
2. To see the results of your mapping function, copy and paste the following code snippet to the MATLAB Command Window.

```matlab
modelName = 'slexAutotransRootInportsExample';
Throttlex = timeseries(zeros(10,1));
Brake     = timeseries(ones(10,1));
signalNames= {'Throttlex', 'Brake'};
signals    = { Throttlex, Brake };
open_system(modelName);
inputMap   = AlmostBlockName(modelName, signalNames, signals);
inputStr   = getInputString(inputMap,'base');
close_system(modelName);
```

After running the code snippet, the variable `inputStr` contains the string 'Throttlex,Brake'.

If your signals are in a Simulink.SimulationData.Dataset, to see the results of your mapping function, use the following code snippet at the MATLAB Command Window.

```matlab
modelName = 'slexAutotransRootInportsExample';
Throttlex = timeseries(zeros(10,1));
Brake     = timeseries(ones(10,1));
ds        = Simulink.SimulationData.Dataset;
ds        = ds.addElement( Throttlex, 'Throttlex' );
ds        = ds.addElement( Brake, 'Brake' );
signalNames= {'ds'};
signals    = { ds }; 
open_system(modelName);
inputMap   = AlmostBlockName(modelName, signalNames, signals);
inputStr   = getInputString(inputMap,'base');
close_system(modelName);
```
After running the code snippet for signals in a Simulink.SimulationData.Dataset, the variable `inputStr` contains the string

'ds.getElement('Throttlex'),ds.getElement('Brake')'.
Converting Harness-Driven Models to Use Harness-Free External Inputs

This example shows how to convert a harness model that uses a Signal Builder block as an input to a harness-free model with root imports. The example collects data from the harness model and stores it in MAT-files for the harness-free model. After storing the data, the example removes the Signal Builder block from the harness model and adds root imports to create a harness-free model. Then, the data in the MAT-files is mapped to the root imports of the model.

Save Harness Data to MAT-Files

Before converting the model to be harness-free, collect the test cases in the harness.

For this example, you will modify the model sldemo_autotrans from the Modeling an Automatic Transmission Controller example.

Open the example model. In the MATLAB Command Window, type sldemo_autotrans.

Modeling an Automatic Transmission Controller

Double-click on ManeuversGUI and select a maneuver
**Exporting Signal Builder Block Groups**

Export data that defines Signal Builder block signal groups to a MAT-file from the Signal Builder window. To export Signal Builder signal data, formatted as Simulink.SimulationData.Dataset, to a MAT-file, open the Signal Builder window and select **File > Export Data > To MAT-File**. In the dialog, enter a name for the MAT-file to contain the data and the number of the group you want to export. For this example, the file name is `slexAutotransRootInportPassingManeuver.mat` and the group number is 1 for the Passing Maneuver group.

**Remove the Signal Builder Block**

Remove the Signal Builder block named ManeuversGUI and replace it with two inports.

1. Delete the Signal Builder block named ManeuversGUI.
2. From the Simulink/Commonly Used Blocks library, drag two inport blocks into the model.
3. Connect the input ports to the lines previously connected to the Signal Builder block.
4. Rename the inport ports. Name the input port connected to the Throttle line **Throttle**. Name the input port connected to the BrakeTorque line **Brake**.

Save the model as `slexAutotransRootInportsExample1.slx` or use the example `slexAutotransRootInportsExample.slx`.

The remaining steps of this example use the model `slexAutotransRootInportsExample.slx`. If you saved the model with a different name use your model name in the steps going forward.
Set Up Harness-Free Inputs

Now that the model is harness-free, set up the inputs that you already saved (See "Save Harness Data to MAT-Files").

In the Modeling tab, select Model Settings. In the Data Import/Export pane, click the Connect Input button.

Map Signals to Root Inport

The Root Inport Mapper tool opens.

The example uses this tool to set up the model inputs from the MAT-file and map those inputs to an input port, based on a mapping algorithm. To select the MAT-file that contains the input data, click the From MAT-File button on the Root Inport Mapper.
toolbar. When the link dialog appears, click the Browse button. In the browser, select the MAT-file that you saved earlier.

Select a Mapping Mode

When you select the MAT-file slexAutotransRootInportPassingManeuver.mat that contains the input data, determine the root input port to which to send input data. Simulink matches input data with input ports based on one of five criteria:

- **Port Order** - Maps in the order it appears in the file to the corresponding port number.
- **Block Name** - Maps by variable name to the corresponding root inport with the matching block name.
• **Signal Name** - Maps by variable name to the corresponding root inport with the matching signal name.

• **Block Path** - Maps by the BlockPath parameter to the corresponding root inport with the matching block path.

• **Custom** - Maps using a MATLAB function.

Earlier in this example, you saved input data to variables of the same name as the harness signals Throttle and Brake, and added input ports with names matching the variables. Given the set of conditions for the input data and the model input ports, the best choice for a mapping criteria is **Block Name**. Using this criteria, Simulink tries to match input data variable names to the names of the input ports. To select this option:

1. Click the **Block Name** radio button.
2. Click the **Options** button and select **Update Model**. This verifies the mapping.
3. Click the **Map** button.

When compiling the data, Simulink evaluates inports against the following criteria to determine compatibility issues. The status of this compatibility is reflected by the table colors green, orange, or red. Warnings and errors are flagged with diagnostic messages. If the **Options > Update Model** option is not selected, Root Inport Mapper determines the compatibility status by evaluating these block parameters and assigned signals:

• **Data Type** - Double, single, enum, ....

• **Complexity** - Real or complex

• **Dimensions** - Signal dimensions vs port dimensions
Finalize the Inputs to the Model

Review the results of the mapping compatibility. Click the Scenario Dataset 'PassingManeuver' in the scenario dataset list. To prepare for simulation, click **Mark for Simulation**. This action applies the mapping variables to the Configuration Parameter Data Import/Export > External Input text box. If this text box has content, it is overwritten.

Simulating the Model

With the changes applied you can now simulate the model and view the results. Run the model. To view the results of the simulation, double-click the Scope Block **PlotResults**.
Modeling an Automatic Transmission Controller with Root Inports
Logging States in Structure Format

This example shows how logging state trajectories of a Simulink® model in a structure format can be better than the traditional method of logging states in an array format. The ordering of the states along the columns in the logged matrix depends on the block sorted order, which the Simulink software determines during model compilation. Various factors can affect the block sorted order, which can alter the ordering of the states.

This example illustrates how logging the states in the Structure format, which stores the block names with the state trajectories, can help prevent the state ordering problem.

Problem with Logging in Array Format

By default, the Simulink software logs the state trajectories in array format, which is a matrix with N columns, where N is the number of states. The matrix has M rows, with each row corresponding to a single simulation time step. This M-by-N matrix form is easy to manipulate in MATLAB®. However, the ordering of the state variables along the columns of the logged matrix depends on the block sorted order. Therefore, any MATLAB code which expects a fixed mapping between the states of blocks in the model and the columns of the states matrix is prone to breaking when the block sorted order changes due to changes in the model.

For example, consider the following two block diagrams:

```matlab
mdl1 = 'sldemo_state_logging1';
mdl2 = 'sldemo_state_logging2';
open_system(mdl1);
open_system(mdl2);
```

![Block diagram 1](image1)

![Block diagram 2](image2)
The two diagrams have the same blocks, the only difference is the ordering of output ports. Simulate the models and log the states in array format:

```matlab
simOut1 = sim(mdl1, 'SaveFormat','Array');
simOut2 = sim(mdl2, 'SaveFormat','Array');
```

Extract the states vectors from the Simulink.SimulationOutput object, which contains the output logs of the simulation:

```matlab
x1 = simOut1.get('xout');
x2 = simOut2.get('xout');
```

Note that the relative ordering of the integrator blocks is different in the two block diagrams. This causes the logged states x1 and x2 to differ because the mapping between the columns and the states is different:

```matlab
isequal(x1, x2)
```

```matlab
ans =
    logical
   0
```

**Using Structure Format Logging**

Simulate the models again, but this time log the states in structure format:
simOut1 = sim(mdl1, 'SaveFormat', 'Structure');
simOut2 = sim(mdl2, 'SaveFormat', 'Structure');

Extract the structures, which contain the state logs, from the simulation output object:

x1s = simOut1.get('xout');
x2s = simOut2.get('xout');

Display these structures. Note that these structures have two fields: time and signals. The field 'time' is empty because we chose 'Structure' for the model parameter 'SaveFormat'. We could have chosen 'StructureWithTime' to store the time vector in the states structure:

disp(x1s);
disp(x2s);

    time: []
    signals: [1x2 struct]

    time: []
    signals: [1x2 struct]

The software logs the state trajectories into `xs.signals(k).values` along with the names of blocks `xs.signals(k).blockName` that correspond to these states. Extract the states into a matrix (like in array format) like this:

x1a = [x1s.signals.values];
x2a = [x2s.signals.values];

Note that the state ordering problem still exists (x1a and x2a are the same as x1 and x2, obtained via array format):

isequal(x1a, x2a)

ans =

    logical

    0
Obtaining States Matrix with Fixed State Order

To fix the state ordering problem, use the block names stored along with the values to map the states to a fixed order, for example, the alphabetical order of the block names:

\[
[~, \text{idx1}] = \text{sort}({\text{x1s.signals.blockName}});
\text{x1} = [{\text{x1s.signals(idx1).values}}];

[~, \text{idx2}] = \text{sort}({\text{x2s.signals.blockName}});
\text{x2} = [{\text{x2s.signals(idx2).values}}];
\]

\text{isequal(x1, x2)}

\[
\text{ans} = \\
\text{logical} \\
1
\]

By re-ordering the signals arrays in \text{x1} and \text{x2} into the alphabetical order of the block names, and extracting the values fields, in that order, into the matrices \text{x1} and \text{x2}, we have a mechanism for logging the states into a matrix with a fixed mapping of the block states to columns of the logged matrix.

Close the models and clear the variables which were used in this example:

\text{close_system(mdl1)}; \\
\text{close_system(mdl2)}; \\
\text{clear ans idx1 idx2 mdl1 mdl2 simOut1 simOut2 x1 x1a x1s x2 x2a x2s}
Saving and Restoring Simulation Operating Point

This example shows the workflow and the benefits of saving and restoring a simulation operating point using a ModelOperatingPoint object.

The ModelOperatingPoint object contains the complete snapshot of all variables that are related to the simulation of a model. After saving the ModelOperatingPoint object at the end of a simulation, Simulink® can reload the operating point and continue the simulation from the time at which the ModelOperatingPoint object was saved. This action produces the same simulation results as if the simulation were never interrupted.

This example illustrates:

• How saving the Final states (logged states) is not always sufficient for complete and accurate restoration of a simulation state.

• How you can save and restore the complete operating point of a simulation, yielding faster simulation workflows

Difficulties with Models with Variable Transport Delay

The states that are logged during a simulation are a subset of all the information needed to fully describe the state of a simulation. Certain blocks rely on internal information that is not logged as part of state logging and final state export. One such block is the Transport Delay block. Models with Transport Delay blocks are usually difficult to fully restore to a particular state because the state of the transport delay is not saved in the structure format or the array format as part of "Final states" data logging.

To illustrate this issue, compare the simulation results for two cases:

1. Simulate a model, which contains a Transport Delay block, from 0 to 5 seconds and save the "Final states" values in the workspace. Then load this first set of final states and simulate from 5 to 10 seconds.

2. Simulate the same model from 0 to 10 seconds and force the model to produce an output at 5 seconds. We call the result of this simulation the baseline result because it is a nonstop simulation.

The results of the first simulation match the first half of the baseline result. If the simulation state of the first simulation had been restored completely, the second simulation results would match the second half of the baseline.
To begin, load this model:

```matlab
mdl = 'slexVariableTransportDelay';
load_system(mdl);
```

Simulate until time of 5 and save the final state in structure format:

```matlab
out = sim(mdl, 'StopTime', '5', 'SaveFinalState', 'on', ...
          'FinalStateName','xFinal', 'SaveFormat','Structure');
y1 = out.ScopeData;
```

Load the final state from the last simulation and run to 10:

```matlab
assignin('base', 'xFinal', out.get('xFinal'));
out1 = sim(mdl, 'StartTime', '5', 'StopTime', '10', ...
          'SaveFinalState', 'off', ...
          'LoadInitialState', 'on', 'InitialState', 'xFinal');
y2 = out1.ScopeData;
```

Run a nonstop simulation which will serve as the baseline result:

```matlab
out2 = sim(mdl, 'OutputOption', 'AdditionalOutputTimes' ,...
          'OutputTimes',[0 5 10], 'LoadInitialState', 'off');
y = out2.ScopeData;
```

Plot the results. Note that the second half of the baseline result does not match the simulation from 5 to 10 seconds, whose initial state was restored from the final state saved at 5 seconds:

```matlab
figure;
for idx=1:3
    subplot(3, 1, idx);
    plot(y.time,y.signals(idx).values);
    hold on;
    plot([y1.time; y2.time],...
         [y1.signals(idx).values;y2.signals(idx).values],'r--');
    hold off;
    grid on;
end
```
Saving and Restoring the Complete Operating Point of a Simulation

You can save the complete final operating point in a Simulink.op.ModelOperatingPoint object. The ModelOperatingPoint object contains all the variables which are needed to restore the simulation results. By using the complete ModelOperatingPoint, Simulink is able to restore the simulation state completely and to reproduce the baseline simulation results.

Set the parameter for Simulink to save the complete operating point at the end of the simulation.

```matlab
out3 = sim(mdl, 'StopTime', '5', 'SaveFinalState', 'on', ...
          'LoadInitialState', 'off', 'SaveOperatingPoint', 'on',...
```

15-461
Load the ModelOperatingPoint from the last simulation and run for an additional 5 seconds. The start time value must remain 0.0 (that value was the start time of the original simulation). The software stores the original simulation start time in xFinal. This value must match the start time of the current simulation to enable restoration of the simulation state.

```matlab
assignin('base', 'xFinal', out3.get('xFinal'));
out4 = sim(mdl, 'StopTime', '10', 'SaveFinalState', 'off', ...
     'LoadInitialState', 'on', 'InitialState', 'xFinal');
```

Plot the results and compare them with the baseline simulation. Note that this time, the simulation state has been completely restored and the simulation results after the operating point restore match the baseline.

```matlab
figure;
for idx=1:3
    subplot(3, 1, idx);
    plot(y.time,y.signals(idx).values);
    hold on
    plot([y1.time; y2.time],...
         [y1.signals(idx).values;y2.signals(idx).values],'r--');
    hold off;
    grid on;
end
```
Close the model and clear the variables which were used in this example

close_system(mdl, 0);
clear mdl idx xFinal y y1 y2 y3 out out1 out2 out3 out4
Logging Intervals

This example shows how to specify logging intervals. Simulink limits the data logging to the specified intervals.

Open the Example Model

Open the example model.

Description of the Example

In this example, you can reduce the amount of logged data by limiting data logging to specific simulation-time intervals. The Increment block in this example has two signals: Increment and Reset. Double-click the Increment block to see the signals. In this example, you will specify two intervals to log the data of two system events triggered by
the Increment and Reset signals. The first interval captures the first reset event. The second interval captures the behavior of the system after the increment event takes place and the signals reach the lower saturation limits. The intervals are based on the Increment and Reset signal values.

**Set up the Logging Intervals and Simulate the Model**

You can specify the Logging Intervals parameter value in "**Configuration Parameters > Data Import/Export > Logging intervals**" settings. To enable the Logging Intervals parameter, turn on "Single simulation output". Click here to open the Configuration Parameters Dialog.

The Logging Intervals parameter value must be a real double matrix with two columns. The matrix elements cannot be NaN. Each row defines the start and end times for an interval. Intervals must be disjoint and ordered.

Set Logging Intervals to \([9,10.5;14.5,30]\) and simulate the model.

OUTERDATA is a wide signal with two elements. The scope shows the values of the OUTERDATA signal during simulation. The scope does not honor the Logging Intervals specification. The Logging Intervals specification is honored when logging time, root outports, signals, Datastore memory, states, and other kinds of data. The Figure shows the logged values of the OUTERDATA captured as part of signal logging:

```
>> out.get('topOut').get('OUTERDATA')
```

The MATLAB `stem` function is being used to display the logged data - ignoring the interpolation settings of the data.
Exit

Close the model.
Working with Big Data

This example shows how Simulink models handle big data as input to and output from a simulation.

Open the Example Model

Open the example model.

Description of the Example

Big data refers to data that is too large to load into system memory all at once.

Simulink can produce big data as simulation output and consume big data as simulation input. To handle big data for both input and output, the entire data is stored in a MAT-file.
on the hard disk. Only small chunks of this data are loaded into system memory at any
time during simulation. This approach is known as streaming. Simulink can stream data
to and from a MAT-file. Streaming solves the memory issues because the capacity of the
hard disk is typically much greater than the capacity of the random access memory.

This example shows how to handle big data in Simulink simulations. The logging to file
capability is used to stream big data as the output of a simulation. Streaming from file
then supplies big data as input to a simulation.

**Set up Logging to File**

To stream output data to a MAT-file, enable logging to file.

Enable logging to file by selecting the "Configuration Parameters > Data Import/
Export > Log Dataset data to file" settings checkbox. Click here to open the
Configuration Parameters Dialog. You can also specify the name of the file that will
contain the result.

The programmatic way to enable Logging to file is by setting the model parameter
LoggingToFile to on.

When logging to file is enabled on a model, simulation of that model streams logged
signals directly into the MAT-file. Additionally, if logging of States or Output is enabled
and SaveFormat is specified as Dataset, those values are streamed into same the MAT-file.

**Simulate the Model**

This example changes the directory to a temporary directory, which has write
permissions. Then the example calls the sim command to simulate the model, logging to
file.

Set the parameter SignalLoggingName, which specifies the name of the Dataset object to
hold the result of signal logging, to topOut. Set the parameter LoggingFileName,
which specifies the name of the resulting MAT-file, to top.mat. The StopTime parameter
is set to 5000 seconds. For a more realistic big data example, the stop time would be a
much larger value, which would result in many more data samples to log.

**Create a DatasetRef Object to Reference the Logged Dataset Within the MAT-file**

Use a DatasetRef object to reference the resulting Dataset in the logged MAT-file. The
benefit of using DatasetRef is that the referenced MAT-file is not loaded into memory.
DatasetRef is a very light wrapper object for referencing a Dataset that is stored in a file.
The alternative of calling the `load` function on this file loads the entire file into memory, which might not be possible if this Dataset contains big data.

**Obtain a Reference to a Logged Signal**

You can use `{ }` indexing of DatasetRef object to reference individual signals within a Dataset, without loading these signals into memory. For example, to reference the seconds signal:

The Values field of `sig2` is a SimulationDatastore object, which is a light-weight reference to the data of signal 2, stored on disk:

```matlab
ans =

SimulationDatastore with properties:
    ReadSize: 100
    NumSamples: 50001
    FileName: 'C:\TEMP\Bdoc19b_1192687_6748\ibF7BE2B\31\top.mat'

Data Preview:
  Time  Data
    ____  ______
    0 sec  1    5
    0.1 sec  1    5
    0.2 sec  2    6
    0.3 sec  2    6
    0.4 sec  3    7
    :      :      :
```

**Obtain More References to Other Logged Signals**

This example uses some of these logged signals as inputs to the simulation of the referenced model. Create light-weight references for each of these. These are bus signals in the model and the resulting Values fields are structures of SimulationDatastore objects. Each structure reflects the hierarchy of the original bus signal.

**Create a New Dataset Object to Use as Simulation Input**

Specify the input signals to a simulation through a Dataset object. Each element in this Dataset provides input data to the inport block corresponding to the same index. Create
an empty Dataset `ds` and then place the references to the logged signals into it as elements number one and two.

Use `{ }` indexing on the Dataset object to assign elements into appropriate positions.

Within each element of the Dataset, you can mix references to signal data (e.g., SimulationDatastore object) with in-memory data (e.g., timeseries objects). To change one of the upper saturation limits from 30 to 37:

**Stream Input Data into Simulation**

Now simulate the referenced model `sldemo_mdlref_counter_bus`, and use the Dataset `ds` as input. The data that is referenced by SimulationDatastore objects is streamed into the simulation without overwhelming the system.

The data for upper saturation limit is not streamed because that signal is specified as an in-memory timeseries. The change in saturation limit is reflected at around time 6 in the scope (the signal now saturates to a value of 37 instead of 30).
Summary

This example has demonstrated a round trip workflow of big data from and to simulation. Logging to persistent storage was used to stream data from the first simulation into a MAT-file. A second simulation was then set up to stream the data from that file as input. A more realistic example would have a larger value for the model StopTime parameter, resulting in a larger logged MAT-file. The second simulation could also be configured for a longer StopTime. However, even with the larger data files for output and input, the memory requirements for the longer simulations remain the same!

MATLAB Workflow

SimulationDatastore allows you to analyze the logged data incrementally in MATLAB. Going back to the reference to the second logged signal, assign the datastore to a new variable to simplify access to it.

Access the Data in Chunks

SimulationDatastore allows incremental reading of the referenced data. The reading is done in chunks and is controlled by the ReadSize property. The default value for ReadSize
is 100 samples (each sample for a signal is the data logged for a single time step of simulation). Change it to 1000 for this example. Each read of the datastore returns a timetable representation of the data.

**Reset the SimulationDatastore Read Counter**

Each read on the datastore advances the read counter. You can reset this counter and start reading from the beginning:

**Iterate Through All Data in the Datastore**

Use SimulationDatastore for incremental access to the logged simulation data for big data analysis in MATLAB. You can iterate over the entire data record and chunks:

**Exit**

Close the model.

**Related Documentation**

Click here for more information on the SimulationDatastore class.
Simulink Subsystem Semantics

This set of examples shows different types of Simulink® Subsystems and what semantics are used when simulating these Subsystems. Each example provides a description of the model and the subtleties governing how it will be executed.

Examples are provided for the following Subsystem types.

- Virtual and Nonvirtual Subsystems
- Function-call Subsystems
- Triggered Subsystems
- Enabled Subsystems
- Triggered and Enabled Subsystems
- If/else Subsystems
- While Subsystems
- For Subsystems
Subsystem Semantics

- Virtual and nonvirtual subsystem overview
- Function-call subsystems
- If and switch-case action subsystems
- Enabled with trigger subsystems
- Triggered subsystems
- Enabled subsystems
- Resettable subsystems
- For Each subsystems
- While subsystems
- For subsystems

Copyright 1990-2015 The MathWorks, Inc.
If-Then-Else Blocks

This example shows what happens when a sine wave is fed into an If block. After running the simulation, the scope shows three plots. This example is designed to compare the If block with enabled subsystems.

The first of these plots shows the original sine wave and the absolute value of the original sine wave offset by two. The absolute value of the sine wave is only shown when the subsystem is executed. This occurs between 0 and 2.5 seconds and 5 and 7.5 seconds. When the subsystem is not executed then the output of the subsystem is reset to 0. The value of 0 is determined by the initial condition of the outport. You can see when the subsystem is executed by looking at the second plot. A value of 1 indicates that it is executed.

The second plot shows the original value of the input signal. If the signal is towards the top of the plot then the if subsystem related to the first plot is executed. If the signal is towards the bottom of the plot then the else subsystem related to the third plot is executed.

The third plot shows the original sine wave and the saturation limited sine wave offset by two. The saturated value of the sine wave is only shown when the subsystem is executed. This occurs between 2.5 and 5 seconds and 7.5 and 10 seconds.
Triggered Subsystems

This model shows triggered subsystems and describes the different trigger types. A triggered subsystem is executed for a single time step whenever the trigger port transitions from false to true in a specified direction. The transition of the trigger port may occur when the direction of the transition is rising, falling, or both rising and falling. A rising transition occurs when the trigger signal transitions from zero or below to a positive value. A falling transition occurs when the trigger signal transitions from zero or above to a negative value.

After running the simulation, look closely at the top plot in the scope. This shows a sine wave, a waveform, and the grounded value of zero. As the waveform transitions through zero, each of the subsystems is triggered appropriately. The output of each subsystem is equal to the current value of the sine wave at that time.
Triggered Subsystem Demonstration

Waveform: square
Freq: 1 Hertz

Freq: 8 rad/sec

Copyright 1990-2015 The MathWorks, Inc.
Enabled Subsystems

This example shows what happens when a sine wave is fed into an enabled subsystem. After running the simulation, the scope shows three plots.

The first of these plots shows the original sine wave and the absolute value of the original sine wave offset by two. The absolute value of the sine wave is only shown when the subsystem is enabled. This occurs between 0 and 2.5 seconds and 5 and 7.5 seconds. When the subsystem is not enabled then the output of the subsystem is reset to 0. The value of 0 is determined by the initial condition of the outport. You can see when the subsystem is enabled by looking at the second plot. A value of 1 indicates that it is enabled.

The second plot shows the original value of the enable signal. If the signal is towards the top of the plot then the subsystem related to the first plot is enabled. If the signal is towards the bottom of the plot then the subsystem related to the third plot is enabled. The difference on when the subsystems are enabled is caused by the logical NOT block.

The third plot shows the original sine wave and the saturation limited sine wave offset by two. The saturated value of the sine wave is only shown when the subsystem is enabled. This occurs between 2.5 and 5 seconds and 7.5 and 10 seconds.
Enabled Subsystem Example

Advanced Enabled Subsystems

This model shows what happens when a sine wave is fed into an enabled subsystem. The four subsystems in the model contain integrators, either discrete or continuous as described by the Subsystem name. After running the simulation, the two scopes show the results for the discrete enabled subsystems (top scope) and continuous enabled subsystems (bottom scope.)

Each Scope shows the output for all combinations of where the states or outputs of the enabled subsystem are either held or reset when the subsystem is disabled. The colors on the scope correspond to the following state and output configuration.

Yellow: Reset states; Held outputs
Purple: Reset states; Reset outputs
Cyan: Held states; Held outputs
Red: Held states; Reset outputs
Resettable Subsystems

This example shows what happens when a ramp signal is integrated using two resettable subsystems. After running the simulation, the scope shows three plots.

The middle plot shows the original value of the reset signal. In this example, we are using a discrete sample time pulse signal. The rising and falling edges are what cause the two subsystems to reset.

The top and bottom plots show the original ramp as a reference and its integration. The subsystems corresponding to the top and bottom plots reset on every rising and falling edge of the pulse, respectively. When each subsystem resets, the state of the integrator block is set to its initial condition value, which is 0. You can refer to the middle plot to determine when each subsystem resets.

Notice how the reset happens for blocks within the subsystem, without the need for each block to have its own reset port. This provides a more convenient way of handling reset at a subsystem level.
Discrete and Continuous Resettable Subsystems

This example shows the comparison between behaviors of discrete and continuous resettable subsystems. After running the simulation, the two scopes show the results for the discrete resettable subsystem (top scope), and continuous enabled subsystems (bottom scope).
Continuous Time Outputs

- Yellow: Continuous Reset Signal
- Blue: Integration of Const
- Red: Integration of Sine

Time range: 0 to 10
Sample rate: 10,000
Variant Subsystems

This model illustrates Simulink® variant subsystems. Variant subsystems let you provide multiple implementations for a subsystem where only one implementation is active during simulation. You can programmatically swap out the active implementation and replace it with one of the other implementations without modifying the model.

Overview of Variant Subsystems

A Variant Subsystem block contains two or more child subsystems where one child is active during model execution. The active child subsystem is referred to as the active variant. You can programmatically switch the active variant of the Variant Subsystem block by changing values of variables in the base workspace, or by manually overriding variant selection using the Variant Subsystem block dialog. The active variant is programmatically wired to the Inport and Outport blocks of the Variant Subsystem by Simulink during model compilation.

To programmatically control variant selection, a Simulink.Variant object is associated with each child subsystem in the Variant Subsystem block dialog. Simulink.Variant objects are created in the MATLAB® base workspace. These objects have a property named Condition, which is an expression, that evaluates to a boolean value and is used to determine the active variant child subsystem. For example, defining

\[ VSS\_LINEAR\_CONTROLLER=\text{Simulink.Variant('VSS\_MODE==1');} \]

in the base workspace creates a Simulink.Variant object where the constructor argument ('VSS_MODE==1') defines when the variant is active. Using the Variant Subsystem dialog, you then associate VSS_LINEAR_CONTROLLER with one of the child subsystems within the Variant Subsystem. Defining

\[ VSS\_MODE=1 \]

in the base workspace, activates the VSS_LINEAR_CONTROLLER variant. The condition argument can be a simple expression consisting of scalar variables, enumerations, equality, inequality, &&, , and ~. Parenthesis () can be used for precedence grouping.

Using Variant Subsystems

The model in this example uses the following variant objects and variant control variable, which are defined in the MATLAB base workspace:

\[ VSS\_LINEAR\_CONTROLLER=\text{Simulink.Variant('VSS\_MODE==1');} \]
VSS_NONLINEAR_CONTROLLER=Simulink.Variant('VSS_Mode==2');

VSS_MODE=2;

Opening the example model sldemo_variant_subsystems runs the **PreLoadFcn** defined in File -> ModelProperties -> Callbacks. This populates the base workspace with the variables for the Variant Subsystem block named Controller:

![Variant Subsystems Diagram](image)

**Figure 1**: The example model, sldemo_variant_subsystems

To specify the Simulink.Variant objects association for the Controller subsystem, right-click on the Controller subsystem and select **Subsystem Parameters**, which will open the Controller subsystem block dialog.

The Controller subsystem block dialog specifies two potential variants. The two variants are in turn associated with the two Simulink.Variant objects **VSS_LINEAR_CONTROLLER** and **VSS_NONLINEAR_CONTROLLER**, which exist in the base workspace. These objects have a property named **Condition**, an expression that evaluates to a boolean and that determines which variant is active. The condition is also shown in the Variant Subsystem block dialog. In this example, the Condition properties of **VSS_LINEAR_CONTROLLER** and **VSS_NONLINEAR_CONTROLLER** are **VSS_MODE == 1** and **VSS_MODE == 2**, respectively. The variable **VSS_MODE** resides in the base workspace, and can be a standard MATLAB variable or a Simulink.Parameter.
If there is no associated variant object or a '%' (comment) character prefixes the variant object in the Variant Subsystem parameters dialog box, then the child subsystem is considered commented out and is not used during model execution.

**Figure 2:** Contents of the Controller subsystem block

Within a Variant Subsystem block, you can place Inport, Outport, and Subsystem blocks. In this example, the Linear Controller Subsystem block is associated with the variant object, VSS_LINEAR_CONTROLLER, and the Nonlinear Controller Subsystem block is associated with the variant object, VSS_NONLINEAR_CONTROLLER.

Signal connections are not allowed in the Variant Subsystem. Simulink programmatically wires up the Inport and Outport blocks to the active variant when simulating the model.

**Switching Active Variants**

To simulate using the Linear Controller variant, define:

\[ \text{VSS\_MODE}=1 \]

in the base workspace and then simulate the model.

**Figure 3:** Simulation using the Linear Controller variant

To simulate using the Nonlinear Controller, define

\[ \text{VSS\_MODE}=2 \]

in the base workspace and then simulate the model.

**Figure 4:** Simulation using the Nonlinear Controller variant
Enumerations and Reuse

The `sldemo_variant_subsystems_enum` model illustrates the following Simulink.Variant capabilities:

1. **Enumerations**: MATLAB enumeration classes can be used to improve readability in the conditions of the variant object.

2. **Reuse**: Simulink.Variant objects can be reused in different Variant Subsystem blocks.

This example uses the following variables which are defined in the MATLAB base workspace:

```matlab
VSSE_LINEAR_CONTROLLER=Simulink.Variant( ... 'VSSE_MODE==sldemo_vss_CONTROLLER_TYPE.LINEAR')

VSSE_NONLINEAR_CONTROLLER=Simulink.Variant( ... 'VSSE_MODE==sldemo_vss_CONTROLLER_TYPE.NONLINEAR')

VSSE_MODE=sldemo_vss_CONTROLLER_TYPE.LINEAR

VSSE_PROTOTYPE=Simulink.Variant( ... 'VSSE_MODE_BUILD==sldemo_vss_BUILD_TYPE.PROTOTYPE')

VSSE_PRODUCTION=Simulink.Variant( ... 'VSSE_MODE_BUILD==sldemo_vss_BUILD_TYPE.PRODUCTION')

VSSE_MODE_BUILD=sldemo_vss_BUILD_TYPE.PRODUCTION
```

In these Simulink.Variant objects, we use the enumeration classes, `sldemo_vss_BUILD_TYPE.m` and `sldemo_vss_CONTROLLER_TYPE.m` to define the Simulink.Variant **Condition** parameters which improves readability.

The three filter Variant Subsystems blocks, Filter1, Filter2, and Filter3 all use the `VSSE_PROTOTYPE` and `VSSE_PRODUCTION` Simulink.Variant objects.

Opening the example model `sldemo_variant_subsystems_enum` runs the **PreLoadFcn** defined in File -> ModelProperties -> Callbacks. This populates the base workspace with variables for the Variant Subsystem blocks:
Figure 5: The example model, sldemo_variant_subsystems_enum

Code Generation

You can use the Simulink® Coder™ to generate code from a model containing Variant Subsystem blocks. By default the generated code contains only the active variant. Alternatively, you can generate code for all variants guarded by C preprocessor conditionals (#if, #elif, #endif) when using the Embedded Coder™.

To generate preprocessor conditionals, the types of blocks that you can place within the child subsystems of a Variant Subsystem block are limited. During the code generation process, one Merge block is placed at the input of each Outport block within the variant.
subsystem and connected to the child subsystems within the variant subsystem. Thus, the restrictions placed on Merge blocks apply to the contents of Variant Subsystem blocks. The restriction checks are only performed when generating code. In addition, the child subsystems of the Variant Subsystem block must be Atomic subsystems, which are created by selecting the **Treat as atomic unit** parameter of the Subsystem parameters dialog box.

Code generation of preprocessor conditionals is active when

1. Embedded Coder target is selected on the Code Generation pane of the Configuration Parameters dialog box.
2. The **Override variant conditions and use following variant** is *not* selected on the Variant Subsystem block parameter dialog box.

When code generation of preprocessor conditionals is active, the generated code contains all child subsystems of the Variant Subsystem blocks protected by C preprocessor conditionals. In this case, the selection of the active variants is deferred until compile-time of the generated code. Only one variant object which is encoded in C macros, must evaluate to true (be active).

In addition, the variant control variables (such as VSS_MODE and VSSE_MODE above) must be Simulink.Parameter objects that specify how the #define's for the variant control variables are managed in the generated code. For the sldemo_variant_subsystems, you can clear the VSS_MODE double from the base workspace. Use Model Explorer -> Simulink Root -> Base Workspace to add a new Simulink.Parameter named VSS_MODE. Specify the parameter **Value** as 1 or 2, the **Data Type** as int32, and **Storage class** as ImportedDefine with **Custom attributes Header file** as sldemo_variant_subsystems_vdef.h. You should then create this header in the current working directory:

```c
/* File: sldemo_variant_subsystems_vdef.h */
#ifndef SLDEMO_VARIANT_SUBSYSTEM_VDEF_H
#define SLDEMO_VARIANT_SUBSYSTEM_VDEF_H
#define VSS_MODE 1
#endif
```

See the Embedded Coder documentation for more information on code generation for variant subsystems.
Block Priority

This model shows what happens when blocks are assigned different priorities. The block priority affects the order in which the blocks are executed. You can set the block priority through the Block Properties dialog.

The block sorting order is calculated only at the beginning of a simulation. For this reason, changes to a block's Priority property are only updated when the simulation is started. Double click on the green block to change the priority levels for the blocks and see what happens.

Inside of each triggered subsystem is an S-Function which controls the coloring of the blocks and also slows down the simulation. The priority level of the block is displayed using the format attribute string that can be set in the Block Properties dialog.
Monitoring Ink Status on a Shared Printer Using Simulink Functions

This example shows how clients, in this case three computers, can send jobs to a server, a printer, and receive status from that server. This example highlights how Simulink Functions can be called from different subsystems and interact both in Simulink® and Stateflow®.

Example Requirements

Note that when you simulate the example, Stateflow might generate code in the current working folder.

Overview

In this example, the computers each generate randomly sized print jobs at regular intervals. The printer interacts with the computers through two Simulink Functions on the Simulink canvas and one Exported Graphical Function in Stateflow.

1. The clients use Function Caller blocks to invoke the addPrintJob interface of the printer.

2. The client-server interaction, modeled by this call, may have a negative return value if the printer has run out of ink.

The example shows how to interface Simulink and Stateflow using functions.

1. The implementation of addPrintJob uses a Function Caller block to invoke queuePrintJob, which is a Stateflow Exported Graphical Function.

2. The queuePrintJob function interacts with other content in the chart using a local chart variable 'work'.

3. The implementation of the chart makes a call to printerInk, which is a Simulink Function.

4. The printerInk function job uses the graphical input and outputs to interact with the addPrintJob function.

The example also shows how to visualize and debug Simulink Functions.
1. You can enable the visualization lines connecting functions and their respective callers. In the Simulink Editor, on the Debug tab, under Information Overlays, click Function Connectors.

2. The Message Viewer block shows where calls to functions are generated, and which functions are called at different times. This helps to understand the flow of calls with respect to time, and also analyze the arguments sent during these calls to functions.
Modeling Services and Access from Application Software

This examples shows a typical software modeling pattern involving services and access to those services in an application.

Overview

Driver model represents services provided, read() and write(), related to motion detection.

The chart in the Software model makes calls to the services using the notation `slexServicesDriverMdlRef.write()`. The notation compactly identifies the function to be called, and the Driver model from where the function’s argument attributes are inherited, making this Software model independently compilable.

Simulating the model, and inspecting Sequence Viewer block shows how the chart makes motion detection decisions based on the services.
Closing the Models
Modeling Reusable Components Using Multiply Instanced Simulink Functions

In this example, the model slexCounterFunctionMdlRef contains a Simulink Function update() which maintains a counter. Multiple instancing provides the ability to reuse this logic as CounterA and CounterB in this application model.

**Overview**

The qualified calling notation CounterA.update and CounterB.update is used to distinguish the two different instances of the counter logic. Note that each instance may be called from multiple caller blocks as illustrated in this example.
Closing the Models
Dynamic Priority Scheduling of Functions

This example shows a common modeling pattern to dynamically schedule the execution of functions using a Stateflow chart.

Overview

Simulink Messages provides intrinsic support for creating and managing priority queues which may be used to model the situation in which the execution order of the functions may change between execution intervals.

Additionally, the Sequence Viewer allows you to visualize execution sequences. To view the dynamically changing execution sequence: 1. Double-click on the 'Sequence Viewer' block 2. Click the 'Run' button once.

Closing the Models
Component-Based Modeling with Model Reference

This example shows how to use Model Reference. It walks you through simulation and code generation of a model that references another model multiple times.

Example Requirements

During this example, Simulink® and Simulink® Coder™ generate code in the code generation folder created in the current working directory. If you do not want to (or cannot) generate files in this directory, you should change your working directory.

Simulink Coder is required to generate code to be deployed in standalone applications.

If you plan to alter the example models:

1. Preserve the example in its original state by copying the following files from your MATLAB® installation directory, without changing their names, to a different directory:
   toolbox/simulink/simdemos/simfeatures/modelreference/sldemo_mdlref_basic.slx
   toolbox/simulink/simdemos/simfeatures/modelreference/sldemo_mdlref_counter.slx

2. Change your current working directory to the directory to which you copied the files

3. Continue with the example

About Model Reference

Open the example model sldemo_mdlref_basic. This model contains three Model blocks: CounterA, CounterB and CounterC. These blocks reference the same model, sldemo_mdlref_counter, which is a separate model and not a subsystem of sldemo_mdlref_basic.

Simulation Mode

The Model block has a parameter named SimulationMode that controls how the referenced model is simulated. If the parameter is set to Accelerator, the referenced model is simulated through code generation. If the parameter is set to Normal, the referenced model is simulated in interpreted mode. In this model, the Model blocks CounterA and CounterB reference sldemo_mdlref_counter in Normal mode, which is indicated by the hollow corners on the Model block icon. The other instances, CounterC, references sldemo_mdlref_counter in Accelerator mode, which is indicated by the filled corners on the Model block icon. To learn more about interacting with models that
are referenced more than once in Normal mode, see the example sldemo_mdlref_depgraph.

Figure 1: The top model, sldemo_mdlref_basic
Figure 2: The referenced model, sldemo_mdlref_counter

Model Reference has several advantages over subsystems:

- You can develop the referenced model independently from the models in which it is used.
- You can reference a model multiple times in another model without having to make redundant copies.
- Multiple models can reference a single model.
- The referenced model is not loaded until it is needed, a feature called Incremental Loading. Incremental loading speeds up model load times.
- If a model is referenced in Accelerator mode, Simulink creates special binaries to be used in simulations. If the binaries are up-to-date, because the referenced model has not changed since the binaries were created, no code generation occurs when models...
that use these binaries are simulated or compiled. This is called *Incremental Code Generation*. The use of binaries increases simulation speed and modularity in code generation.

- Generating code for a model with Model blocks also takes advantage of incremental code generation.

**Simulation Through Code Generation (Does not require Simulink Coder)**

Simulation of a model that references other models in Accelerator mode requires the generation of code and a binary file called the Simulation Target for each unique referenced model. This process does not require Simulink Coder. For this example, a simulation target is built for the `sldemo.mdlref_counter` model. You can create the Simulation Target by any of the following actions:

- Updating `sldemo.mdlref_basic`
- Simulating `sldemo.mdlref_basic`
- Issuing the command:

  ```
  slbuild('sldemo.mdlref_counter','ModelReferenceSimTarget')
  ```

**Remarks:**

- If a model is referenced in Normal mode, the simulation target is not built for that referenced model.
- Once the simulation target is built, subsequently simulating or updating the model does not trigger a rebuild of the referenced model's simulation target unless the referenced model has changed.
- Turn on the sample time colors for `sldemo.mdlref_basic`. Note that the Model blocks have inherited different sample times. The Model Blocks inherit their sample times if the referenced model does not explicitly specify a sample time.

**Code Generation for Standalone Applications (Requires Simulink Coder)**

When creating a standalone executable for `sldemo.mdlref_basic`, the code and binaries for the targets of `sldemo.mdlref_counter` are first generated. The necessary targets are built if they are not up-to-date. You can build the target for `sldemo.mdlref_counter` and the standalone executable for `sldemo.mdlref_basic` by any of the following actions:

- Building the standalone executable for `sldemo.mdlref_basic`. 
• Building the Model Reference target of `sldemo_mdlref_counter`, followed by building the standalone executable for `sldemo_mdlref_basic`.

Remarks:
• Once the model reference coder target of `sldemo_mdlref_counter` is built, subsequent builds of `sldemo_mdlref_basic` do not trigger a rebuild of `sldemo_mdlref_counter` unless `sldemo_mdlref_counter` changes (Incremental Code Generation).
• Note that the code generated for the referenced model `sldemo_mdlref_counter` is reused.
• The HTML report of the generated code for `sldemo_mdlref_basic` has a link to the HTML report of the generated code for `sldemo_mdlref_counter` in the Referenced Model section.

Build the Standalone Target and view the HTML report of the generated code

Export Simulation Data
You can log time, outputs, states, and final states data of the simulation. The logging is configured to use Dataset format.

Incremental Loading
Close `sldemo_mdlref_counter` and `sldemo_mdlref_basic` and then re-open `sldemo_mdlref_basic`. At the MATLAB prompt issue the command:

```matlab
find_system('type','block_diagram')
```

The referenced model is not listed because it is not loaded. Double click on any Model block. At the MATLAB prompt, issue the command:

```matlab
find_system('type','block_diagram')
```

The referenced model is now listed, illustrating how incremental loading works.
Viewing Signals in Model Reference Instances

This example shows how to use the Simulation Data Inspector to view and analyze signals in referenced models. The Simulation Data Inspector relies on signal logging so this example shows how to configure your model to log signals in referenced models.

Open the Example Models

Run the following command to create and open a working copy of the project files.

```
slexMultiInstanceModelStart
```
The project example copies files to your temporary folder so that you can edit them and use them under local version control.

The Simulink® Project Tool opens and loads the project already under version control.
The project is configured to run some startup tasks.

**Setting up Signals to Log in Referenced Models**

The first task in setting up a referenced model to view signals using the Simulation Data Inspector is to set up that model for logging. To do that, edit the referenced model and mark the signals for logging. Note that four signals are already set up for logging (marked with the blue wi-fi badge). To log an additional signal:

1. Select the raw output signal.
2. In the **Simulation** tab, from the **Prepare** gallery, click **Log Signals** as shown below.
Once the signals to log have been selected in the referenced model, the next step is to select which signals to actually log from the topmost model in the hierarchy. In the top model, on the **Modeling** tab, click **Model Settings**.
On the **Data Import/Export** pane, click the **Configure Signals to Log...** button to activate the Simulink Signal Logging Selector dialog.
There are two Logging modes that can be used when logging signals in referenced models. The first is Log all signals as specified in model, which will honor any logging settings that were made in referenced models. If you need to override any of those settings, such as selecting only a subset of logged signals, then you should use the Override signals mode. In this mode, you are given the ability to select a subset of loggable signals and configure their properties, such as decimation and name.

**Viewing and Analyzing Logged Signals Using the Simulation Data Inspector**

Once logging has been configured for a model hierarchy and all changes to the models have been saved, the model can be simulated. The simulation creates a Dataset object in the base workspace. You can analyze and view the logged data in this object using
standard MATLAB tools. In addition, Simulink provides the Simulation Data Inspector, for viewing and analyzing data created from simulations. See the documentation for more details on using the Simulation Data Inspector.

With the Simulation Data Inspector, you can:

- View signal traces for any logged signals
- Visually compare signal values for any logged signals
- Compare signal values collected over multiple simulations

After a simulation is done or while using the Simulation Stepper, you will see a highlight around the Data Inspector icon in the Simulation tab, which indicates that clicking it will open the Simulation Data Inspector.

For example, you can use the Simulation Data Inspector to view the output signal of two of the instances of the model LimitedCounter. In the Simulation Data Inspector, you can see that CounterA increases more rapidly than CounterB. This behavior is expected because the Pulse Generator driving this model is running at a faster rate.
You can configure what to display in the inspect table by clicking the gear icon at the right top of the table.

Another useful tool that the Simulation Data Inspector provides is the ability to compare multiple simulation runs. This functionality can help you understand how changes to your model will affect results. For example, you can change the value of the upper limit of the counter models to 8 and simulate the model again to see how that affects output values. To change the value, double click on the block `MultiInstanceModelExample/upper` and modify the Value parameter to 8.

Now if you compare the results from the first simulation for the output of the `CounterA` instance of `LimitedCounter`, you see that the upper limit of this signal has changed from 10 to 8, as expected. Because the tolerances are set to zero, the red octagon next to each signal indicates that the values have changed between runs.
Visualizing Model Reference Architectures

This example shows how to view dependencies among referenced models.

**Example Requirements**

In this example, Simulink® generates files in the current working directory. If you do not want to generate files in this directory, you should change the working directory to a suitable directory before starting this example.

**Open The Example Model**

Open the example model. The model `sldemo_mdlref_depgraph` looks like this:
**Find Referenced Models**

Use the `find_mdlrefs` utility function to find all Model blocks and all models that a model references. Type `help find_mdlrefs` for more information about this function.

```matlab
[refMdls, modelBlks] = find_mdlrefs('sldemo_mdlref_depgraph');
```

`find_mdlrefs` returns two cell arrays, `refMdls` and `modelBlks`. `refMdls` contains the names of all models that are directly or indirectly referenced by `sldemo_mdlref_depgraph`. The last element in `refMdls` is always the name of the input model. `modelBlks` contains block paths for all Model blocks in the top model and all referenced models.

**Viewing a Model Reference Dependency Graph**

To view the dependency graph for the example model, double click on the blue box in the example model. Perform one of these actions to open the dependency viewer for a model:

* Use the `depview` function. * In the **Modeling** tab, select **Compare > Dependencies > Models Only**.

**Dependency Graph**

The next figure shows the dependency graph of `sldemo_mdlref_depgraph`: 

---

15-526
In the dependency graph, the gray nodes represent Simulink models. The directed lines indicate dependencies. For example, in the example model the link from `sldemo_mdlref_depgraph` to `sldemo_mdlref_house` indicates that `sldemo_mdlref_depgraph` references `sldemo_mdlref_house`. If the model had library links, you could select **User-defined Libraries** from the toolstrip to have the dependency viewer show libraries linked to by models in the hierarchy. When **User-defined Libraries** is selected, purple nodes are used to depict libraries.
Interacting with the Dependency Viewer

- To select a node, left-click on it.
- To open the model or library associated with a node, double-click on it.
- To pan the dependency graph, click and hold the left mouse button on an empty space then drag the mouse.
- To zoom into and out of the region under the mouse pointer, use the mouse wheel.
- To center the dependency graph and adjust the zoom so that the dependency graph fills the available space, press the space bar.

Please refer to the documentation for more information about all the features of the Dependency Viewer.

Viewing the Referenced Model Instances

In the previous dependency graph, only one node exists for each model in the graph and at most one link exists from one node to another node. Therefore, this dependency graph does not show if multiple references exist from one model to another model. Furthermore, this view does not show which models are referenced in Normal mode and which models are referenced in Accelerator mode.

The dependency viewer can show referenced model instances, and this view can be accessed in one of two ways:

- In the model, on the Modeling tab, select Compare > Dependencies > Referenced Model Instances.
- From the Model Dependency Viewer, select Model Instances from the toolstrip.

Note that in this view, known as Instance View, libraries are never shown.
Dependency Graph for Referenced Model Instances

In Instance View, the gray nodes represent Simulink models. The directed lines indicate dependencies. Note: in this view two nodes are labeled sldemo_mdlref_F2C because this model is referenced twice, once by sldemo_mdlref_outdoor_temp and once by sldemo_mdlref_heater. Except for the top model, all nodes have corners that denote the simulation mode of the referenced model. White corners denote a Normal mode reference and black corners denote an Accelerator mode reference. Thus,
sldemo_mdlref_heater makes a Normal mode reference to sldemo_mdlref_F2C and an Accelerator mode reference to sldemo_mdlref_thermostat.

**Interacting with the Dependency Viewer in Instance View**

Currently, the reference sldemo_mdlref_outdoor_temp makes to sldemo_mdlref_F2C is an Accelerator mode reference. Suppose you want to change this reference to be a Normal mode reference:

1. Select the node sldemo_mdlref_F2C that sldemo_mdlref_outdoor_temp points to.
2. Click **Highlight Block**. This opens sldemo_mdlref_outdoor_temp and highlights the corresponding Model block.
3. Select the highlighted block.
4. In the **Model Block** tab, change the **Simulation Mode** to Normal.
5. To see this change, go to the Model Dependency Viewer and click **Refresh**.

The Instance View dependency graph now looks like this:
The dependency viewer now shows the reference `sldemo_mdlref_outdoor_temp` makes to `sldemo_mdlref_F2C` as a Normal mode reference, but it has a round info badge on it. This badge appears because `sldemo_mdlref_F2C` is configured to run in Normal mode but its parent, `sldemo_mdlref_outdoor_temp`, is configured to run in Accelerator mode. Normal mode references from models running in Accelerator mode are not supported, so `sldemo_mdlref_F2C` will run in Accelerator mode during simulation.
To make this model run in Normal mode, you must configure all of its ancestors to run in Normal mode. To do this:

1. Select the node sldemo.mdlref.outdoor_temp.
2. Click Highlight Block. This opens sldemo.mdlref.depgraph and highlights the corresponding Model block.
3. Select the highlighted block.
4. In the Model Block tab, change the Simulation Mode to Normal. (If there were more Accelerator mode ancestors of sldemo.mdlref_F2C, these would also need to be changed to Normal mode references.)

To see this change, go to the Model Dependency Viewer and click Refresh.
Interacting with Models That Are Referenced Multiple Times in Normal Mode

There are now two models that reference sldemo_mdlref_F2C in Normal mode. After simulating this model, sl_demo_mdlref_heater looks like this:
And `sldemo_mdlref_outdoor_temp` looks like this:

The Model block that references `sldemo_mdlref_F2C` in `sldemo_mdlref_heater` has white corners, while the Model block that references `sldemo_mdlref_F2C` in `sldemo_mdlref_outdoor_temp` has gray corners. This is because Model block Normal Mode Visibility is on for the Model block in `sldemo_mdlref_heater` but is not on for the Model block in `sldemo_mdlref_outdoor_temp`. Note that only one Normal mode instance of each model will have white corners, the rest will have gray corners. Display blocks (for example, Scopes) will show data for the instance referenced from the Model block with white corners, in this case from the model `sldemo_mdlref_heater`, but not for any other instance:
To open the Model Block Normal Mode Visibility dialog box, from the top model (sldemo_mdlref_depgraph) in the Simulation tab, from the Prepare gallery, click Normal Mode Visibility. Select the instance of each model that is referenced more than once in Normal mode for which you want to have Normal Mode Visibility on.
Right now, no Model blocks have been selected, which means that during the last simulation, Simulink selected one instance of `sldemo_mdlref_F2C` to have Normal Mode Visibility set to on, which was indicated by the Model block with white corners. To select the instance referenced by `sldemo_mdlref_outdoor_temp`, check the check box corresponding to that instance, then press **OK**:

After changing the Normal Mode Visibility setting, simulate the model `sldemo_mdlref_depgraph` to make use of the new setting.
Visualizing Model Reference Architectures

Fahrenheit to Celsius

Copyright 1990-2014 The MathWorks, Inc.
Introduction to Managing Data with Model Reference

This example shows the basic concepts related to managing data with model reference.

For a more detailed discussion of this topic, see: “Detailed Workflow for Managing Data with Model Reference” on page 15-553.

NOTE: This example uses the same model as the more detailed example. As a result, the model uses certain features that this introductory example does not describe.

Example Requirements

During this example, Simulink® generates code in the code generation folder created in the current directory. If you do not want to (or if you cannot) generate files in this directory, you should change your working directory.

If you plan to alter the example models:

1. Preserve the example in its original state by copying the following files from your MATLAB® installation directory, without changing their names, to a different directory:

   toolbox/simulink/simdemos/simfeatures/modelreference/sldemo_mdlref_datamngt.slx
   toolbox/simulink/simdemos/simfeatures/modelreference/sldemo_mdlref_datamngt_wsdata.m
   toolbox/simulink/simdemos/simfeatures/modelreference/sldemo_mdlref_counter_datamngt.slx
   toolbox/simulink/simdemos/simfeatures/modelreference/sldemo_mdlref_counter_datamngt_types.m
   toolbox/simulink/simdemos/simfeatures/modelreference/sldemo_mdlref_counter_datamngt_wsdata.m

2. Change your current directory to the directory to which you copied the files.

3. Continue with the example.

Open the Example Model

open_system('sldemo_mdlref_datamngt')
Example Content

This example uses a top model (sldemo.mdlref_datamngt) that contains three Model blocks: Counter1, Counter2, and Counter3. These blocks reference the same model (sldemo.mdlref_counter_datamngt).

The referenced model implements a limited counter algorithm that:

- Resets the counter if the first trigger input changes
- Increments the counter by a specified amount if the second input changes
- Saturates the counter between the specified upper and lower limits
The referenced model outputs a bus signal that contains:

- **Count**: the value of the counter as an 8-bit integer
- **OverflowState**: an enumerated value that indicates whether the counter is at the upper limit, lower limit, or in range

**Example Outline**

**The sequence of steps in the example is as follows:**

1. Prepare the referenced model (sldemo_mdlref_counter_datamngt) to use structures for the signals and parameters in its external interface.

2. Prepare the top model (sldemo_mdlref_datamngt) to call the referenced model.

3. Simulate the top model and examine the results.

**Referenced Model: Setting Up Parameters**

The referenced model has two model arguments (CounterParams, CounterICs) that parameterize the blocks in the model. Model arguments provide different parameter values to each instance of a referenced model. In this model, the arguments are defined as parameter structures to reduce the number of arguments being passed to the referenced model.

**The process for defining the model arguments is as follows:**

1. Open the referenced model (sldemo_mdlref_counter_datamngt).
2. Define MATLAB structures to parameterize the referenced model.

```matlab
CounterParams.Increment = int8(1);
CounterParams.LowerLimit = int8(-10);
CounterParams.UpperLimit = int8(10);

CounterICs.Count = int8(0);
CounterICs.OverflowState = SlDemoRangeCheck.InRange;
```

**NOTE:** This example uses an 8-bit integer for the counter, so the numeric fields also use 8-bit integers.

3. Use the MATLAB structures as model arguments.

Model arguments are defined as variables in the model workspace of a referenced model. You can initialize the model workspace from a number of different data sources. This example uses a MATLAB script (`sldemo_mdlref_counter_datamngt_wsdata`) to create the parameter structures for the model arguments. Using a MATLAB script makes it easy to create and modify the parameter structure outside the model. It also facilitates incremental changes, version control, and data differencing.
Set the data source in the dialog of the model workspace.

You can use the Model Explorer to view and edit the contents of the model workspace.

Select the Argument checkbox to configure variables as model arguments.

Referenced Model: Defining the Shape of the Output Bus Signal

The referenced model produces two results and packages them into a bus signal:
• **Count**: the value of the counter as an 8-bit integer
• **OverflowState**: an enumerated value that indicates whether the counter is at the upper limit, lower limit, or in range

**To define the bus type for the root output of the referenced model:**

1. Use the Bus Editor to define the bus object (OutputType).

![Bus Editor - Manage Bus Objects in the Base Workspace](image)

Alternatively, you can create the bus object at the MATLAB command line:

```matlab
OutputType = Simulink.Bus;  
OutputType.Elements = Simulink.BusElement;  
OutputType.Elements(1).Name = 'Count';  
OutputType.Elements(1).DataType = 'int8';  
OutputType.Elements(2) = Simulink.BusElement;  
OutputType.Elements(2).Name = 'OverflowState';  
OutputType.Elements(2).DataType = 'Enum: SlDemoRangeCheck';
```

2. Configure the root outport of the referenced model to output a nonvirtual bus signal based on this bus object (OutputType).
Referenced Model: Setting Initial Value for Bus Output

In general, the initial values for bus signals and states can be specified as '0', in which case all of the elements of the bus will be initialized to zero (or the relevant ground value). However, in certain cases, it is desirable to specify nonzero initial values for bus signals and states. In this example, the initial condition of the counter is tunable, so the initial value of output signal must be set consistently.

To specify the initial value of the output of the referenced model:
1. Create a parameter structure that is compatible with the bus signal that you want to initialize. One of the model arguments (CounterICs) is a parameter structure that matches the shape of the output signal. This structure was defined as follows:

```plaintext
CounterICs.Count = int8(0);
CounterICs.OverflowState = SlDemoRangeCheck.InRange;
```

2. Open the dialog box for the root outport and enter the name of the structure (CounterICs) into the **Initial output** field.
NOTE: In general, when initializing a bus signal or state, the parameter structure does not need to match the bus type exactly, but its fields must be a subset of the elements in the bus object and the attributes of these fields must match the elements in the bus object.

Referenced Model: Masking Referenced Model

It is often useful to mask the referenced model to customize the user interface. You can create a mask on a model and then reference it from a Model block. To mask a model, click Create Model Mask in the model workspace section of Model Explorer for the model.
When you reference a masked model from a Model block, a mask is generated automatically on the Model block. For more information, see the topic Create and Reference a Masked Model.
Top Model: Parameterizing Model Blocks

The referenced model is configured to accept a structure for its model argument. This example calls each instance of the referenced model with different parameter values.

1. Open the top model (sldemo_mdlref_datamngt).

2. Create parameter structures with the same "shape" as the model arguments defined in the referenced model.

   \[
   \text{Param1.Increment} = \text{int8}(1);
   \text{Param1.LowerLimit} = \text{int8}(-20);
   \text{Param1.UpperLimit} = \text{int8}(20);
   \]
Param2 = Param1;
Param2.Increment = int8(2);

IC1.Count = int8(0);
IC1.OverflowState = SlDemoRangeCheck.InRange;

IC2 = IC1;
IC2.Count = int8(-10);

3. Set the mask parameters on the masked Model blocks to use these parameters.

For example, the parameters of the first Model block ('sldemo_mdlref_datamngt/Counter1') are set as:

- **Param1** for the **Counter parameters**
- **IC1** for the **Initial conditions**
Top Model: Scheduling Calls to the Referenced Model

As mentioned earlier, the referenced model implements a limited counter algorithm with two trigger inputs. The algorithm detects "rising edges" in the trigger inputs and reacts as follows:

- Resets the counter if the first trigger input changes.
- Increments the counter by a specified amount if the second input changes.

In this example, you drive all three instances of the counter algorithm with the same trigger inputs, generated by the Stimulus subsystem. The period and sample time of the
trigger inputs are defined by the values entered for the mask parameters of the Stimulus subsystem:

- Reset counter every 4 seconds.
- Increment counter 5 times per second (period = 0.2 second).
- Sample time is 0.1 second.

**NOTE:** Reset and increment periods must be at least 2 times the sample time.

### Simulating the Top Model

1. Save or close the referenced model.

You need to save the referenced model before it can be used by the top model. Alternatively, you can close the referenced model and use the original version of the model provided with this example.

2. Simulate the top model by selecting **Simulation > Run**.

   \[ \text{sim('sldemo_mdlref_datamngt')} \]

**NOTE:** Simulating the top model automatically generates a simulation target for the referenced model.

3. Review the simulation results as displayed by the Scope blocks.
Detailed Workflow for Managing Data with Model Reference

This example shows a number of tools and techniques that make up a workflow for managing data with model reference.

For a basic introduction to this topic, see: “Introduction to Managing Data with Model Reference” on page 15-538.

Example Requirements

During this example, Simulink® and Simulink® Coder™ generate code in the code generation folder created in the current directory. If you do not want to (or if you cannot) generate files in this directory, you should change your working directory. Simulink Coder is required to generate model reference binaries to be deployed in standalone applications.

If you plan to alter the example models:

1. Preserve the example in its original state by copying the following files from your MATLAB® installation directory, without changing their names, to a different directory:

   toolbox/simulink/simdemos/simfeatures/modelreference/sldemo_mdlref_datamngt.slx
   toolbox/simulink/simdemos/simfeatures/modelreference/sldemo_mdlref_datamngt_wsdata.m
   toolbox/simulink/simdemos/simfeatures/modelreference/sldemo_mdlref_counter_datamngt.slx
   toolbox/simulink/simdemos/simfeatures/modelreference/sldemo_mdlref_counter_datamngt_types.m
   toolbox/simulink/simdemos/simfeatures/modelreference/sldemo_mdlref_counter_datamngt_wsdata.m

2. Change your current directory to the directory to which you copied the files.

3. Continue with the example.

Open the Example Model

open_system('sldemo_mdlref_datamngt')
Example Content

This example uses a top model (sldemo_mdlref_datamngt) that contains three Model blocks: Counter1, Counter2, and Counter3. These blocks reference the same model (sldemo_mdlref_counter_datamngt).

The referenced model implements a limited counter algorithm that:

- Resets the counter if the first trigger input changes
- Increments the counter by a specified amount if the second input changes
- Saturates the counter between the specified upper and lower limits
The referenced model outputs a bus signal that contains:

- **Count**: the value of the counter as an 8-bit integer
- **OverflowState**: an enumerated value that indicates whether the counter is at the upper limit, lower limit, or in range

This model is configured to show a number of Simulink features working together:

- **Tunable parameter structures**: Collect variables into a MATLAB structure that parameterizes the model. In this example, the model arguments of the referenced model are defined as parameter structures.
- **Model arguments**: Pass different parameter values to each model reference instance.
- **Bus objects**: Define the "shape" of structures for signals and parameters used in the model reference interface. Define type name for structure parameters and signals in generated code.
- **Simulink.Bus.createMATLABStruct**: Static method for creating a MATLAB structure that matches the shape of the bus object.
- **Bus initialization**: Uses a parameter structure to initialize bus signals and states.
- **Masked Model block**: Creates a customized interface for a model reference block.
- **Triggered model reference**: Explicit control over scheduling of components.

This example also uses other features that are useful for managing data in Simulink:

- **Simulink.saveVars**: Serialize workspace variables to a MATLAB script. Supports incremental modification, data differencing, and version control.
- **Simulink.findVars**: Discover how workspace variables are used by a model.

**Example Outline**

**The sequence of steps in the example is as follows:**

1. Prepare the referenced model (`sldemo_mdlref_counter_datamngt`) to use structures for the signals and parameters in its external interface.

2. Prepare the top model (`sldemo_mdlref_datamngt`) to call the referenced model.

3. Simulate the top model and examine the results.

4. Generate code to create a standalone executable for the top model.
Referenced Model: Setting Up Parameters

The referenced model has two model arguments (CounterParams, CounterICs) that parameterize the blocks in the model. Model arguments provide different parameter values to each instance of a referenced model. In this model, the arguments are defined as parameter structures to reduce the number of arguments being passed to the referenced model.

The process for defining the model arguments is as follows:

1. Open the referenced model (sldemo_mdlref_counter_datamngt).

2. Define MATLAB structures to parameterize the referenced model.

   CounterParams.Increment = int8(1);
   CounterParams.LowerLimit = int8(-10);
   CounterParams.UpperLimit = int8(10);

   CounterICs.Count = int8(0);
   CounterICs.OverflowState = SlDemoRangeCheck.InRange;
NOTE: This example uses an 8-bit integer for the counter, so the numeric fields also use 8-bit integers.

3. Use the MATLAB structures as model arguments.

Model arguments are defined as variables in the model workspace of a referenced model. You can initialize the model workspace from a number of different data sources. This example uses a MATLAB script (`sldemo_mdlref_counter_datamngt_wsdata`) to create the parameter structures for the model arguments. Using a MATLAB script makes it easy to create and modify the parameter structure outside the model. It also facilitates incremental changes, version control, and data differencing.

Set the data source in the dialog of the model workspace.

Alternatively, you can set the data source from the MATLAB command line:

```matlab
hWS = get_param('sldemo_mdlref_counter_datamngt', 'ModelWorkspace');
hWS.DataSource = 'MATLAB File';
hWS.FileName = 'sldemo_mdlref_counter_datamngt_wsdata.m';
hWS.reload;
```

You can use the Model Explorer to view and edit the contents of the model workspace.

Select the Argument checkbox to configure variables as model arguments.
Alternatively, you can set the list of model arguments from the MATLAB command line:

```matlab
set_param('sldemo_mdlref_counter_datamngt', ...
    'ParameterArgumentNames', 'CounterParams,CounterICs');
```

4. Explore how the referenced model uses the model arguments.

A number of blocks in the referenced model use the model arguments:

- A Constant block specifies the increment amount (CounterParams.Increment).
- Various blocks inside the "Range Check" subsystem use the values of the lower and upper limits (CounterParams.LowerLimit and CounterParams.UpperLimit).
- Various blocks use the initial condition of the counter (CounterICs.Count).
- The root Outport block uses the initial condition structure (CounterICs) to initialize the system output prior to execution.

You can use `Simulink.findVars` to discover the blocks that use the model argument.

```matlab
paramInfo = Simulink.findVars('sldemo_mdlref_counter_datamngt', ..., ...
    'Name', 'CounterParams');
icInfo    = Simulink.findVars('sldemo_mdlref_counter_datamngt', ..., ...
'SearchMethod', 'cached', ...
'Name', 'CounterICs');

paramBlks = paramInfo.UsedByBlocks
icBlks    = icInfo.UsedByBlocks

paramBlks =

3x1 cell array

{ 'sldemo_mdlref_counter_datamngt/Increment' }
{ 'sldemo_mdlref_counter_datamngt/Range Check/Detect Overflow' }
{ 'sldemo_mdlref_counter_datamngt/Range Check/Saturate Count' }

icBlks =

3x1 cell array

{ 'sldemo_mdlref_counter_datamngt/Initial Count' }
{ 'sldemo_mdlref_counter_datamngt/Previous Count' }
{ 'sldemo_mdlref_counter_datamngt/outputs' }

NOTE: Once you have compiled the model, you can use Simulink.findVars to retrieve the 'cached' variable usage information.

Referenced Model: Defining the Shape of the Output Bus Signal

The referenced model produces two results and packages them into a bus signal:

• Count: the value of the counter as an 8-bit integer
• OverflowState: an enumerated value that indicates whether the counter is at the upper limit, lower limit, or in range

To define the bus type for the root output of the referenced model:

1. Use the Bus Editor to define the bus object (OutputType).
Alternatively, you can create the bus object at the MATLAB command line:

```matlab
OutputType = Simulink.Bus;
OutputType.Elements = Simulink.BusElement;
OutputType.Elements(1).Name = 'Count';
OutputType.Elements(1).DataType = 'int8';
OutputType.Elements(2).Name = 'OverflowState';
OutputType.Elements(2).DataType = 'Enum: SlDemoRangeCheck';
```

2. Configure the root outport of the referenced model to output a nonvirtual bus signal based on this bus object (OutputType).
Alternatively, you can set the block parameters from the MATLAB command line:

```matlab
set_param('slmodel_MDLref_counter_datamngt/outputs', ...
  'UseBusObject', 'on', ...,
  'BusObject', 'OutputType', ...,
  'BusOutputAsStruct', 'on');
```

3. Create a MATLAB script to recreate this bus object.
This example uses a MATLAB script (`sldemo_mdlref_counter_datamngt_types`) to recreate the bus objects used by the referenced model.

You can create this MATLAB script manually, or use `Simulink.saveVars`:

```matlab
Simulink.saveVars('sldemo_mdlref_counter_datamngt_types.m', 'OutputType');
```

4. Configure the referenced model to create the global data that it uses.

It is common practice for a model to recreate all of the global data and types that it uses. In this example, the referenced model's `PreLoadFcn` is used to execute the MATLAB script that creates the bus objects for the referenced model. The referenced model does not use any other global variables.

To set the model's `PreLoadFcn`, open the Model Properties dialog box from the model's File menu or set it from the MATLAB command line.

```matlab
set_param('sldemo_mdlref_counter_datamngt', ...
    'PreLoadFcn', 'sldemo_mdlref_counter_datamngt_types');
```

5. Explore how the bus type is used in the model hierarchy.

This bus type forms part of the interface for the referenced model and is referred to by blocks in both the top and referenced models. Use `find_mdlrefs` and `Simulink.findVars` to find out about all the places where this bus object is used in the model reference hierarchy.

```matlab
models = find_mdlrefs('sldemo_mdlref_datamngt');
open_system(models);
varInfo = Simulink.findVars(models, 'Name', 'OutputType');
blks = vars.UsedByBlocks
```

```matlab
models =
    2x1 cell array
    {'sldemo_mdlref_counter_datamngt'}
    {'sldemo_mdlref_datamngt' }

blks =
    7x1 cell array
Referenced Model: Setting Initial Value for Bus Output

In general, the initial values for bus signals and states can be specified as '0', in which case all of the elements of the bus will be initialized to zero (or the relevant ground value). However, in certain cases, it is desirable to specify nonzero initial values for bus signals and states. In this example, the initial condition of the counter is tunable, so the initial value of output signal must be set consistently.

To specify the initial value of the output of the referenced model:

1. Create a parameter structure that is compatible with the bus signal that you want to initialize. One of the model arguments (CounterICs) is a parameter structure that matches the shape of the output signal. This structure was defined as follows:

   ```
   CounterICs.Count         = int8(0);
   CounterICs.OverflowState = SlDemoRangeCheck.InRange;
   ```

   Alternatively, you could have used the static method,
   `Simulink.Bus.createMATLABStruct` to create the structure using the ground value of the `OutputType`.

   ```
   CounterICs = Simulink.Bus.createMATLABStruct('OutputType');
   ```

2. Open the dialog box for the root output and enter the name of the structure (CounterICs) into the **Initial output** field.
Alternatively, you can set the block parameters from the MATLAB command line:

```matlab
set_param('sldemo_mdlref_counter_datamngt/outputs', ...
'InitialOutput', 'CounterICs');
```

**NOTE:** In general, when initializing a bus signal or state, the parameter structure does not need to match the bus type exactly, but its fields must be a subset of the elements in the bus object and the attributes of these fields must match the elements in the bus object.
Referenced Model: Masking Referenced Model

It is often useful to mask the referenced model to customize the user interface. You can create a mask on a model and then reference it from a Model block. To mask a model, click **Create Model Mask** in the model workspace section of Model Explorer for the model.
When you reference a masked model from a Model block, a mask is generated automatically on the Model block. For more information, see the topic Create and Reference a Masked Model.
**Top Model: Parameterizing Model Blocks**

The referenced model is configured to accept a structure for its model argument. This example calls each instance of the referenced model with different parameter values.

1. Open the top model (`sldemo_mdlref_datamngt`).

2. Create parameter structures with the same "shape" as the model arguments defined in the referenced model.

   ```
   Param1.Increment = int8(1);  
   Param1.LowerLimit = int8(-20);  
   Param1.UpperLimit = int8(20);  
   ```
Param2 = Param1;
Param2.Increment = int8(2);

IC1.Count = int8(0);
IC1.OverflowState = SlDemoRangeCheck.InRange;

IC2 = IC1;
IC2.Count = int8(-10);

3. Set the mask parameters on the masked Model blocks to use these parameters.

For example, the parameters of the first Model block ('sldemo_mdlref_datamngt/Counter1') are set as:

- **Param1** for the **Counter parameters**
- **IC1** for the **Initial conditions**
Alternatively, you can set the block parameters from the MATLAB command line:

```matlab
set_param('sldemo_mdlref_datamngt/Counter1', ...
    'CounterParams', 'Param1', ...
    'CounterICs', 'IC1');
```

3. Create a MATLAB script to recreate these workspace variables.

This example uses a MATLAB script (`sldemo_mdlref_datamngt_wsdata`) to recreate these workspace variables.

You can create this MATLAB script manually, or use `Simulink.saveVars`:

```matlab
Simulink.saveVars('sldemo_mdlref_datamngt_wsdata.m', 'Param*');
```
4. Configure the top model to create all global variables.

It is good practice for the top model to recreate all of the global data and types used throughout the model reference hierarchy. This configuration avoids loading referenced models when simulating the top model.

In this example, the top model's `PreLoadFcn` is used to execute the MATLAB scripts that create the workspace data for the top model and the types for the referenced model. The referenced model does not use any global data.

```matlab
set_param('sldemo_mdlref_datamngt', ...  
    'PreLoadFcn', ['sldemo_mdlref_datamngt_wsdata; ', ...  
        'sldemo_mdlref_counter_datamngt_types']);
```

**Top Model: Scheduling Calls to the Referenced Model**

As mentioned earlier, the referenced model implements a limited counter algorithm with two trigger inputs. The algorithm detects "rising edges" in the trigger inputs and reacts as follows:

- Resets the counter if the first trigger input changes.
- Increments the counter by a specified amount if the second input changes.

In this example, you drive all three instances of the counter algorithm with the same trigger inputs, generated by the Stimulus subsystem. The period and sample time of the trigger inputs are defined by the values entered for the mask parameters of the Stimulus subsystem:

- Reset counter every 4 seconds.
- Increment counter 5 times per second (period = 0.2 second).
- Sample time is 0.1 second.

**NOTE:** Reset and increment periods must be at least 2 times the sample time.

**Simulating the Top Model**

1. Save or close the referenced model.

You need to save the referenced model before it can be used by the top model. Alternatively, you can close the referenced model and use the original version of the model provided with this example.

2. Simulate the top model by selecting **Simulation > Run**.
sim('sldemo_mdlref_datamngt')

**NOTE:** Simulating the top model automatically generates a simulation target for the referenced model.

3. Review the simulation results as displayed by the Scope blocks.
Generating Code for the Top Model (Requires Simulink Coder)

Generate code and build a standalone executable for the top model by selecting Code > C/C++ Code > Build Model.
rtwbuild('sldemo_mdlref_datamngt')

**NOTE:** When you generate code for the top model, Simulink Coder automatically generates code for the referenced model. Use Embedded Coder™ to generate embedded code for production deployment.

### Controlling the Type Name for the Model Argument

By default, Simulink Coder generates an automatic name for the type of the parameter structures. This name is unique and deterministic, but not easy to recognize. You can control this type name by using Simulink.Parameter objects to define the parameter structures and Simulink.Bus objects to define the type.

1. Define the bus type for the model argument by using the Bus Editor or by typing the following commands at the MATLAB command line:

   ```matlab
   CounterParamType = Simulink.Bus;
   CounterParamType.Elements = Simulink.BusElement;
   CounterParamType.Elements(1).Name = 'Increment';
   CounterParamType.Elements(1).DataType = 'int8';
   CounterParamType.Elements(2) = Simulink.BusElement;
   CounterParamType.Elements(2).Name = 'LowerLimit';
   CounterParamType.Elements(2).DataType = 'int8';
   CounterParamType.Elements(3) = Simulink.BusElement;
   CounterParamType.Elements(3).Name = 'UpperLimit';
   CounterParamType.Elements(3).DataType = 'int8';
   ```

2. Modify the existing MATLAB script to recreate this bus object.

   This bus type must be defined in the base workspace, along with the bus type for the referenced model's output. You can manually modify the MATLAB script that defines the types for the referenced model or use `Simulink.saveVars` to append this variable to the existing MATLAB script.

   ```matlab
   Simulink.saveVars('sldemo_mdlref_counter_datamngt_types.m', 'CounterParamType', '-append');
   ```

3. Create Simulink.Parameters to define the model arguments.

   To associate the bus types with the model arguments, you must use Simulink.Parameter objects in place of the MATLAB structures in the model workspace of the referenced model.

   ```matlab
   CounterParams = Simulink.Parameter;
   CounterParams.Value.Increment = 1;
   ```
CounterParams.Value.LowerLimit = -10;
CounterParams.Value.UpperLimit = 10;
CounterParams.DataType = 'Bus: CounterParamType';

CounterICs = Simulink.Parameter;
CounterICs.Value.Count = 0;
CounterICs.Value.OverflowState = SlDemoRangeCheck.InRange;
CounterICs.DataType = 'Bus: OutputType';

**NOTE:** When you set the data type of a parameter object to be a bus, you can use double-precision values for the numeric fields of the parameter structure. Simulink converts these double-precision values to the appropriate numeric data type during model compilation.

You can manually modify the MATLAB script that defines the model argument or use `Simulink.saveVars` to update the existing MATLAB script.

```matlab
Simulink.saveVars('sldemo_mdlref_counter_datamngt_wsdata.m', '-update');
```

**NOTE:** When you use the `-update` option, you do not need to specify the variables to be written out because `Simulink.saveVars` writes out only the values of variables already defined in the MATLAB script.

5. Reload the contents of the model workspace.

You have modified the data source that is used by the referenced model's workspace. If this model is open, you must reload the contents of the model workspace.

You can reload by using the model workspace's dialog box or at the MATLAB command line.

```matlab
hWS = get_param('sldemo_mdlref_counter_datamngt', 'ModelWorkspace');
hWS.reload;
```

**Controlling Representation of Parameters in the Top Model**

You can use a similar approach to convert the parameter structures in the top model to use Simulink.Parameter objects. This configuration gives you control over the type name and enables you to control how these parameters appear in the generated code.

1. Create Simulink.Parameter objects with the ExportedGlobal storage class to define the parameter structures for the top model as a exported, tunable variables in the generated code.
Param1 = Simulink.Parameter;
Param1.Value.Increment = 1;
Param1.Value.LowerLimit = -10;
Param1.Value.UpperLimit = 10;
Param1.DataType = 'Bus: CounterParamType';
Param1.CoderInfo.StorageClass = 'ExportedGlobal';

Param2 = Param1.copy;
Param2.Value.Increment = 2;

IC1 = Simulink.Parameter;
IC1.Value.Count = 0;
IC1.Value.OverflowState = SlDemoRangeCheck.InRange;
IC1.DataType = 'Bus: OutputType';
IC1.CoderInfo.StorageClass = 'ExportedGlobal';

IC2 = IC1.copy;
IC2.Value.Count = -10;

2. Modify the existing MATLAB script to recreate these workspace variables.

You can manually modify the MATLAB script that defines the parameters for the top model or use Simulink.saveVars to update the existing MATLAB script.

Simulink.saveVars('sldemo_mdlref_datamngt_wsdata.m', '-update');

**NOTE:** When you use the -update option, you do not need to specify the variables to be written out because Simulink.saveVars writes out only the values of variables already defined in the MATLAB script.

**Regenerate Code for the Top Model (Requires Simulink Coder)**

Generate code and build a standalone executable for the top model by selecting Code > C/C++ Code > Build Model.

rtwbuild('sldemo_mdlref_datamngt')
Interface Specification Using Bus Objects

This example shows how to propagate bus signals into referenced models. It also shows how you can simulate the referenced models independently using logged signal data from the parent model. (For a quick introduction to Model Reference, see “Component-Based Modeling with Model Reference” on page 15-510. See Simulink® documentation for more information about configuring signals for data logging.)

Open The Example Model

Open the example model
**Example Requirements**

During this example, Simulink and Simulink® Coder™ may generate code in the code generation folder created in the current working directory. If you do not want to (or cannot) generate files in this directory, you should change the working directory to a suitable directory. Simulink Coder is required to generate code to be deployed in standalone applications.

**Description of Bus Signal Setup for the Example**

This model, sldemo_mdlref_bus contains a Model block, CounterA, that references sldemo_mdlref_counter_bus, a model of a simple counter. A bus signal called COUNTERBUS feeds the data and the saturation limits of the counter into the model. COUNTERBUS contains two signals: the data and LIMITBUS. The data signal is used in counting, and LIMITBUS contains the upper and lower limit values of the counter. The following shows the structure of COUNTERBUS:

* COUNTERBUS (bus signal)
  * data
  * LIMITBUS (bus signal)
    * upper_saturation_limit
    * lower_saturation_limit

The Model block CounterA contains a second input port that feeds a bus signal of type INCREMENTBUS. This signal is used to change the increment and reset the counter. The following shows the structure of INCREMENTBUS:

* INCREMENTBUS (bus signal)
  * increment
  * reset

The example also includes a MAT-file named sldemo_mdlref_counter_bus.mat that creates bus objects that define the elements of COUNTERBUS, LIMITBUS and INCREMENTBUS. The preload function of sldemo_mdlref_bus loads this MAT-file when the model is loaded, causing the bus objects to be created in the MATLAB® workspace.

**Note:** You can view these objects by selecting the **Base Workspace** in the Model Explorer. You can also view the bus structures in the Bus Editor, which you can open by typing `buseditor` at the MATLAB command prompt. The model uses these bus objects to specify the outputs of the following Bus Creator blocks:

- COUNTERBUSCreator1
• COUNTERBUSCreator2
• LIMITBUSCreator1
• LIMITBUSCreator2
• IncrementBusCreator

The Inport `counter_input` of the referenced model `sldemo_mdlref_counter_bus` specifies the COUNTERBUS bus object in its **Data type** field on the **Signal Attributes** tab. The Inport `increment_input` is similarly configured to use the bus object INCREMENTBUS.

**Logging Signal Data**

Now you can simulate `sldemo_mdlref_bus` to see the output in the scope window.

Note that the following signals have been marked for signal logging:

• COUNTERBUS
• INCREMENTBUS
• OUTERDATA
• INNERDATA

After the model simulates, the logged signal is available in the **Base Workspace** in the `topOut` variable.

```

   topOut =

Simulink.SimulationData.Dataset 'topOut' with 4 elements

<table>
<thead>
<tr>
<th>Name</th>
<th>BlockPath</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 [1x1 Signal]</td>
<td>COUNTERBUS  sldemo_mdlref_bus/Concatenate</td>
</tr>
<tr>
<td>2 [1x1 Signal]</td>
<td>OUTERDATA  sldemo_mdlref_bus/CounterA</td>
</tr>
<tr>
<td>3 [1x1 Signal]</td>
<td>INCREMENTBUS sldemo_mdlref_bus/IncrementBusCreator</td>
</tr>
<tr>
<td>4 [1x1 Signal]</td>
<td>INNERDATA  ...erA</td>
</tr>
</tbody>
</table>

- Use braces `{ }` to access, modify, or add elements using index.

```

This model uses **Dataset** format for signal logging. To access Dataset format logged data for a given signal, use the `getElement` method:
>> topOut.getElement('COUNTERBUS')

Simulink.SimulationData.Signal
Package: Simulink.SimulationData

Properties:
   Name: 'COUNTERBUS'
   PropagatedName: ''
   BlockPath: [1x1 Simulink.SimulationData.BlockPath]
   PortType: 'outport'
   PortIndex: 1
   Values: [2x1 struct]

Bus data is logged as a MATLAB structure in the Values field:

>> topOut.getElement('COUNTERBUS').Values

2x1 struct array with fields:

    data
    limits

This structure contains MATLAB timeseries objects for each bus signal:

>> topOut.getElement('COUNTERBUS').Values(1).data
and
>> topOut.getElement('COUNTERBUS').Values(2).data

timeseries

Common Properties:
   Name: 'data'
   Time: [301x1 double]
   TimeInfo: tsdata.timemetadata
   Data: [301x1 int32]
   DataInfo: tsdata.datametadata

timeseries

Common Properties:
   Name: 'data'
   Time: [301x1 double]
   TimeInfo: tsdata.timemetadata
You can also plot a logged data using the `plot` command:

```matlab
>> topOut.getElement('OUTERDATA').Values.plot()
```

Logging Model Reference Signals

When simulating the top model `sldemo_mdlref_bus`, you can select the signals in the reference model `sldemo_mdlref_counter_bus` to log using the Signal Logging Selector. Right-click the Model block `CounterA` in the top model and select `Signals &
Ports > Log Model Reference Signals... to choose which signals from the referenced model to log.

Note that for this model, the logging mode Log all signals as specified in model is selected. This means that all the signals that are logged when simulating sldemo_mdlref_counter_bus standalone are logged when simulating the top model. To change logging for any of these signals or to log a subset of signals:

- Set the logging mode to "Override signals".
- Uncheck the top-level model block on the left hierarchy view to indicate that this model should not use "log as specified in model".
- Use the list on the right side to specify which signals to log when simulating the top model.
- To save these changes, save the top model.

Loading Data

Open the referenced model sldemo_mdlref_counter_bus.

The referenced model sldemo_mdlref_counter_bus has been configured to read the logged signal data through the root Inport blocks. You can see how this is done by opening the Configuration Parameters dialog of the referenced model and selecting the Data Import/Export tab. The Input field is configured to read the logged data from topOut for both input ports. A comma-separated list specifies values for the 2 input ports. The increment_input port uses a structure of timeseries to load data for INCREMENTBUS and the counter_input port uses an array of structure of timeseries to load data for COUNTERBUS.

- topOut.getElement('COUNTERBUS')
- topOut.getElement('INCREMENTBUS')

An easy way you can configure the Input field is by clicking the Connect Input button. This action opens the Root Inport Mapper tool. In this example, this tool uses a mapping algorithm to set up the Input field from logged data in the base workspace.

To select the data to import:

1. Click the From Workspace button in the Root Inport Mapper toolstrip.
2. Select the logged data topOut and click the OK button.

Now that the data is loaded into the Root Inport Mapper tool, you can determine the root input port for which to assign input data. Simulink matches input data with input ports...
based on one of five criteria (block name, block path, signal name, or a custom algorithm). Since topOut was logged using signal names from the model, the best choice for a mapping criteria is **Signal Name**. Using this criteria, Simulink tries to match input data variable names to the names of the input signals.

To select this option:

1. In the Root Inport Mapper hierarchy pane, select scenario dataset topOut.
2. Click the **Signal Name** radio button.
3. From the **Map to Model** button dropdown, click **Map All**.

When mapping data, Simulink evaluates inports against the input data to determine compatibility. The Root Inport Mapper table reflects the status of this compatibility with a green check mark, orange warning triangle, or red error exclamation mark. In this example, the table shows a green status indicating there is no compatibility issue. You can simulate the model with this mapping of input data to inports. The map action also sets the **Input** field in the **Data Import/Export** tab of the Configuration Parameters dialog with the proper comma separated list of inputs. To apply the changes to the model, in the Configuration Parameters dialog, click **Apply**.

You can simulate sldemo_mdlref_counter_bus to see the output in the scope window. After simulation, the scopes from sldemo_mdlref_bus and sldemo_mdlref_counter_bus show the same trace. The signal feeding the scope in the referenced model is also logged. The logged data is available in the MATLAB workspace under the variable subOut. You can verify that the data in topOut.getElement('OUTERDATA') and subOut.getElement('INNERDATA') is the same.

**Code Generation For Standalone Applications (Requires Simulink Coder)**

Create the executable and inspect the code for sldemo_mdlref_bus.

**Exit**

Close sldemo_mdlref_counter_bus and sldemo_mdlref_bus.
Converting Subsystems to Model Reference

This example shows how to automate conversion of a model containing buses to use model reference.

**Example Requirements**

During this example, Simulink® generates files in the current working directory. If you do not want to generate files in this directory, you should change the working directory to a suitable directory.

**Open the Example Model**

Open the example model. This model contains an atomic subsystem, Bus Counter, modeling a simple counter.
Convert a Subsystem to Model Reference

You can use Simulink.SubSystem.convertToModelReference to convert an atomic subsystem to model reference. This utility function converts an atomic subsystem by creating a new model, copying the contents of the subsystem into the model, and reconfiguring the root level Inport and Outport blocks and configuration parameters of the model. Then, based on its input arguments, it either replaces the subsystem block with a Model block that references the new model, or it creates another, temporary model containing a Model block that references the model derived from the subsystem block. For more information see Simulink documentation or type:

help Simulink.SubSystem.convertToModelReference

at the MATLAB® prompt.

Perform the Conversion

You can run the following command to convert the Bus Counter subsystem to a Model block.

Simulink.SubSystem.convertToModelReference(...
    'sldemo_mdlref_conversion/Bus Counter', ...
    'sldemo_bus_counter', ...
    'ReplaceSubsystem', true);
Simulate and Generate Code from the Model

You can update, simulate, and generate code for the model containing the Model block.

Export Simulation Data

You can log time, outputs, states, and final states data of the simulation. The logging is configured to use Dataset format.

Close Models and Clear Workspace

Close the models and clear variables added to the workspace during this example.
Model Reference Conversion Advisor

You can use the Model Reference Conversion Advisor to interactively perform the conversion. To do so, set the 'UseConversionAdvisor' parameter to 'true' when calling Simulink.SubSystem.convertToModelReference.

You can also access the advisor by: # Selecting a Subsystem block. # In the Subsystem Block tab, select Convert > Convert to Model Block.
Using Data Stores Across Multiple Models

This example shows how to use and log local data stores. It also shows how to define, use and log global data stores to share global data among referenced models.

Open the Example Models

Open the example models. The top model `sldemo_mdlref_dsm` references models `sldemo_mdlref_dsm_bot` and `sldemo_mdlref_dsm_bot2`. Logging of data stores is turned on for `sldemo_mdlref_dsm` on the Data Import/Export pane of the Configuration Parameters dialog.

Reference model `sldemo_mdlref_dsm_bot` amplifies the incoming signal by 5 if the signal is positive or by 2 if the signal is negative and writes the value of this signal to local data store `RefSignalVal`. This local data store is configured to log all written values to the workspace after simulation. The logging parameters for this local data store are controlled by the Logging tab of the block parameters dialog.

Reference model `sldemo_mdlref_dsm_bot2` sets the value of a global boolean data store named `ErrorCond` to true if the value of its incoming signal is outside of the range (-0.8, 0.8). The top model also monitors the data store `ErrorCond` and switches between the outputs of the two referenced models, depending on the value of that data store.
Using Data Stores Across Multiple Models

Copyright 1990-2013 The MathWorks, Inc.

Copyright 1990-2013 The MathWorks, Inc.
Define a Global Data Store Using a Simulink® Signal Object

To specify to Simulink that the top model and both of the referenced models use the same memory for the Data Store named ErrorCond in each model, create an instance of a Simulink.Signal object named ErrorCond in the base workspace. Since this object is visible to each model, its scope encompasses the individual models and implies that there
is one global data store of that name that is shared among the models. Note that none of the three models contains a Data Store Memory block with a data store name of ErrorCond. If any model contained such a block, that block would define a data store local to the model, which would shadow the global data store.

Since the object ErrorCond defines a global data store, it must explicitly define the DataType, Complexity, Dimensions, and SamplingMode properties to non-inherited settings. SampleTime and StorageClass may be left as their inherited values, or explicitly set.

The example includes a MAT-file named sldemo_mdlref_dsm_data.mat that contains the Simulink.Signal object ErrorCond. The preload function of the model sldemo_mdlref_dsm loads this MAT-file causing the object to be loaded in the base workspace. You can view this object by selecting the base workspace in the Model Explorer. You can also create this object from the MATLAB® command prompt. The following commands define the object used in this example:

```matlab
ErrorCond = Simulink.Signal;
ErrorCond.Description = 'Use to signal that subsystem output is invalid';
ErrorCond.DataType = 'boolean';
ErrorCond.Complexity = 'real';
ErrorCond.Dimensions = 1;
ErrorCond.SamplingMode='Sample based';
ErrorCond.SampleTime = 0.1;
ErrorCond.LoggingInfo.DataLogging = true;
```

**Logging a Global Data Store**

To log all the values written to the global data store, set the DataLogging property of the LoggingInfo property of the Simulink.Signal object. The LoggingInfo property also allows specification of other logging parameters such as decimation and maximum points.

```matlab
Simulink.LoggingInfo (handle)
DataLogging: 1
NameMode: 0
LoggingName: ''
DecimateData: 0
Decimation: 2
LimitDataPoints: 0
MaxPoints: 5000
```
Simulate the Model

Now you can simulate `sldemo_mdlref_dsm` to see the output. The reference model `sldemo_mdlref_dsm_bot` writes to the data store `ErrorCond` while the top model `sldemo_mdlref_dsm` reads from the data store. The blue line represents the input to `sldemo_mdlref_dsm_bot`; the green line represents the output of the Switch block. Note that the output switches when the magnitude of the input signal falls outside of the range.
**View Logged Results**

Both the global data store ErrorCond and the local data store RefSignalVal are configured to log all written values after simulation. These logged results are stored in the base workspace in the `dsmout` variable.

```
dsmout =
Simulink.SimulationData.Dataset 'dsmout' with 2 elements

<table>
<thead>
<tr>
<th>Name</th>
<th>BlockPath</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 [1x1 DataStoreMemory]</td>
<td>ErrorCond</td>
</tr>
<tr>
<td>2 [1x1 DataStoreMemory]</td>
<td>RefSignalVal</td>
</tr>
</tbody>
</table>

- Use braces {} to access, modify, or add elements using index.

To access the data stored for the local data store, use the `getElement` function to get the correct data store element based on the name `RefSignalVal`.

```
>> dsmout.getElement('RefSignalVal')
```

Simulink.SimulationData.DataStoreMemory
Package: Simulink.SimulationData

Properties:
```
Name: 'RefSignalVal'
BlockPath: [1x1 Simulink.SimulationData.BlockPath]
Scope: 'local'
DSMWriterBlockPaths: [1x2 Simulink.SimulationData.BlockPath]
DSM Writers: [101x1 uint32]
Values: [1x1 timeseries]
```

Data is stored in a `timeseries` within the `Values` field.

```
>> plot(dsmout.getElement('RefSignalVal').Values);
```
To determine which block wrote to the data store at a given time, use the DSMWriters property. This array contains a list of indices into the DSMWriterBlockPaths array. For example, the block path of the Data Store Write block that wrote the 5th value to the data store can be obtained as follows:

```matlab
>> dsm = dsmout.getElement('RefSignalVal');
>> dsm.DSMWriterBlockPaths(dsm.DSMWriters(5))
```

Simulink.SimulationData.BlockPath
Package: Simulink.SimulationData

Block Path:
'sldemo_mdlref_dsm/A'
'sldemo_mdlref_dsm_bot/PositiveSS/DSW'
Use the getBlock method to access block path character vectors from this object.
Model Reference Function-Call

This example shows how to use a model reference block which is explicitly triggered by means of its function-call trigger port.

Example Requirements

During this example, Simulink® generates files in the current working directory. If you do not want to generate files in this directory, you should change the working directory to a suitable directory.

Open Example Model

If you haven't already opened it, open the example model.

Model Referenced Function-Call

This model contains three Model blocks: 500ms Counter, 1 sec Counter and Asynchronous Counter. These blocks reference the same model sldemo_mdlref_fcncall_cntr. The model blocks represent saturating counters which increment their output values by one each time they are explicitly triggered through their function-call trigger ports. The Stateflow® Chart Scheduler calls each Model block at its specified rate.

Simulate the Model

Now you can simulate the model sldemo_mdlref_fcncall and verify the behavior of the counters by examining the results displayed on the scope blocks.
Model Reference Function-Call

Copyright 1990-2014 The MathWorks, Inc.
Close the Model

Close the example model.
Protected Models for Model Reference

This example shows how to create and use a protected model in Normal and Accelerated mode simulations, as well as for code generation.

Example Requirements

Before you can generate protected models, you must have Simulink® Coder™ installed. To generate the read-only view of protected models, you must have Simulink® Report Generator™ installed.

Open the Example Model

Open the example model, sldemo_mdlref_bus.

Figure 1: The example model, sldemo_mdlref_bus
Protect the Referenced Model

This model contains one Model block, CounterA. To protect the model referenced by this block, right-click the Model block and select Subsystem & Model Reference > Create Protected Model for Selected Model Block. A dialog box opens where you can select options for creating a protected model.

![Protected model dialog box](image)

**Figure 2:** Protected model dialog box

Under Allow user of protected model to select Open read-only view of model, Simulate, and Generate code. You have the option of entering passwords for the 3 aforementioned functionalities. The 3 passwords are used to protect the respective
functionalities. Enter separate passwords for the 3 options and click Create. A protected model file, sldemo_mdlref_counter_bus.slxp, is generated in the current working folder. You can use this protected model in Normal and Accelerated mode simulations, as well as for code generation. You can also open a read-only view of the model. The original model file is not required.

**Determine the Variables for a Protected Model**

To simulate a model that references a protected model, you might need to define variables in the base workspace. For example, the protected model, sldemo_mdlref_counter_bus, needs the base workspace variables that specify the buses at the root input and output ports of the model. When you ship a protected model, you must include definitions of the required variables or the model is unusable. The `protect` command generates a protected model and determines the required variables. For example:

```matlab
```

The second output of the `protect` function, `neededVars`, includes base workspace variables you must send to the recipient. The value of `neededVars` is a cell array that contains the names of the base workspace variables required for the protected model. However, the cell array might also contain the names of workspace variables that the model does not need.

Before you share a protected model, edit `neededVars` to delete the names of any variables that the model does not need. Save the required workspace variables into a MAT-file:

```matlab
save('sldemo_mdlref_counter_bus_vars', neededVars{:});
```

**Use the Protected Model**

When the Model block references a protected model, a shield badge is displayed on the block.
Figure 3: Referencing a protected model

Right-click on the shield badge and select Authorize.... Enter the passwords you used while creating the protected model.
Now you can simulate the model in either Normal or Accelerator mode. You can also build the top model by selecting **Code > C/C++ Code > Build Model**.

Double-click the Model block to open the read-only view of the model. Alternatively, right-click on the shield badge and select **Show Webview** to open the read-only view of the model.
Figure 5: The protected model web view

Right-click on the shield badge and select Display Report to open the report
Figure 6: The protected model report
Model Reference Variants

This example shows how to use model reference variants. A Model block is used to reference one Simulink® model from another Simulink model. A Variant Subsystem block can contain Model blocks as variants. A variant describes one of N possible modes a Variant Subsystem block can operate in. Each variant references a specific model with its associated model-specific arguments. Only one variant is active for simulation. You can switch the active variant by changing the values of variables in the base workspace, or by manually overriding variant selection using the Variant Subsystem block dialog.

For new models, use a Variant Subsystem block instead of a Model block to contain model variants, unless you need to use variants that are conditionally executed models (models with control ports). Support for using a Model block to contain model variants will be removed in a future release.

Example Requirements

During this example, Simulink and Simulink® Coder™ may generate code in the code generation folder in the current working folder. If you do not want to (or cannot) generate files in this folder, you should change the working folder.

Overview of Model Variants

A Model block is used to reference one Simulink model (the child model) from another Simulink model (the parent model). A Variant Subsystem block can have Model blocks as variants. The variants comprise a set of models that have the potential to be referenced by the Variant Subsystem block. In this example, there are two models that are potentially referenced by the Variant Subsystem block named Controller.

Each variant is associated with a Simulink.Variant object. Simulink.Variant objects are created in the MATLAB® base workspace. For example:

V_LINEAR_CONTROLLER=Simulink.Variant('CTRL==1');

where the constructor argument ('CTRL==1') defines the condition when the variant is active. In this case, defining:

CTRL=1

in the base workspace would activate the V_LINEAR_CONTROLLER variant. The condition argument can be a simple expression consisting of scalar variables, enumerations, equality, inequality, &&, , and ~. Parenthesis () can be used for precedence grouping.
For a given Variant Subsystem block, one variant is active during simulation. The *active variant* is determined by evaluating the variant object conditions in the MATLAB base workspace. Alternatively, you can instruct the Variant Subsystem block to use a specific variant by selecting the **Override variant conditions and use following variant** checkbox.

**Specifying Variants**

The model used in this example requires the following variables be defined in the MATLAB base workspace:

\[
\begin{align*}
V_{\text{LINEAR\_CONTROLLER}} &= \text{Simulink}.\text{Variant}('CTRL==1'); \\
V_{\text{NONLINEAR\_CONTROLLER}} &= \text{Simulink}.\text{Variant}('CTRL==2'); \\
CTRL &= 2;
\end{align*}
\]

Opening the model `sldemo_mdlref_variants` runs the **PreLoadFcn** defined in File -&gt; ModelProperties -&gt; Callbacks. This callback populates the base workspace with the variables for the Variant Subsystem block named Controller:

![Model Reference Variants](image)

**Figure 1:** The top model, `sldemo_mdlref_variants`

Right-click the Variant Subsystem block Controller and select the menu item **Block Parameters (Subsystem)** to open the block dialog box.
The dialog box specifies two potential variants. The two variants are in turn associated with the two Simulink.Variant objects V_LINEAR_CONTROLLER and V_NONLINEAR_CONTROLLER, which exist in the base workspace. These objects have a property named **Condition**, an expression that evaluates to a boolean and that determines which variant is active. The **Condition** is also shown in the Variant Subsystem block dialog. In this example, the condition of V_LINEAR_CONTROLLER and V_NONLINEAR_CONTROLLER are CTRL == 1 and CTRL == 2, respectively. The variable CTRL resides in the base workspace, and may be a standard MATLAB variable or a Simulink.Parameter.

**Switching Active Variants**

To simulate using the sldemo_mrv_linear_controller, define:

CTRL=1

in the base workspace and then simulate the model.

---

**Figure 2**: Simulation using the sldemo_mrv_linear_controller variant
To simulate using the `sldemo_nonlinear_controller`, define

\[ \text{CTRL}=2 \]

in the base workspace and then simulate the model.

![Simulation result](image)

**Figure 3**: Simulation using the `sldemo_mrv_nonlinear_controller` variant

**Enumerations and Reuse**

The `sldemo_mdlref_variants_enum` model illustrates `Simulink.Variant` capabilities:

1. **Enumerations**: MATLAB enumeration classes can be used to improve readability in the conditions of the variant object.

2. **Reuse**: `Simulink.Variant` objects can be reused in different Variant Subsystem blocks.

This example requires the following variables be defined in the MATLAB base workspace:
VE_LINEAR_CONTROLLER=Simulink.Variant('E_CTRL==sldemo_mrv_CONTROLLER_TYPE.LINEAR')

VE_NONLINEAR_CONTROLLER=Simulink.Variant('E_CTRL==sldemo_mrv_CONTROLLER_TYPE.NONLINEAR')

E_CTRL=sldemo_mrv_CONTROLLER_TYPE.LINEAR

VE_PROTOTYPE=Simulink.Variant('E_CURRENT_BUILD==sldemo_mrv_BUILD_TYPE.PROTOTYPE')

VE_PRODUCTION=Simulink.Variant('E_CURRENT_BUILD==sldemo_mrv_BUILD_TYPE.PRODUCTION')

E_CURRENT_BUILD=sldemo_mrv_BUILD_TYPE.PRODUCTION

In these Simulink.Variant objects we use the enumeration classes sldemo_mrv_BUILD_TYPE.m and sldemo_mrv_CONTROLLER_TYPE.m within the Simulink.Variant Condition properties to improve readability.

The VE_PROTOTYPE and VE_PRODUCTION Simulink.Variant objects are reused across the three filter Variant Subsystem blocks, Filter1, Filter2, and Filter3.

Opening the model sldemo_mdlref_variants_enum runs the PreLoadFcn defined in File -> ModelProperties -> Callbacks. This callback populates the base workspace with variables for the Variant Subsystem blocks, which are displayed in the MATLAB Command Window:
Assigning Tasks to Cores for Multicore Programming

This example shows how to take advantage of executing code on a multicore processor by graphical partitioning. This example requires Simulink Coder™ to generate multithreaded code.

Introduction

Multicore programming allows you to make use of the processing power of modern multicore processors to create high-performance applications. Simulink™ allows you to take advantage of multicore programming by allowing you to graphically partition your algorithms and to assign the code generated from those partitions to parallel threads on your multicore processor.

The parallel threads are typically scheduled by an operating system, assigning threads to cores dynamically. This allows the scheduler to make good use of the cores while providing a notion of fairness. However, in many applications, you may need additional control on how the cores are used or you may want to dedicate a set of cores to specific logic, for example to improve determinism. This level of control is achievable through graphical partitioning and is illustrated in this example.

Try the Example

A typical situation for multicore consists of a multi-rate model for which you want to assign code for the fast rates to one core and the code for the slower rates to other cores. This may allow you to add more logic to these faster rates or to achieve more determinism. More generally, you may want to dedicate cores to application logic with the greatest demand on resources. To begin, open the multicore project:

slexMulticoreExample
Generate Multithreaded Code

This application model is already partitioned into four concurrent tasks through the Concurrent Execution dialog which you may access from the Solvers pane of the model's Configuration Parameters. Use this dialog to specify logic, task and core assignments. In this example, we have partitioned the application logic as:
Double-click on the 'Generate Code and Profile Report' button to generate multi-threaded code, and profile its execution. This action runs and instruments the generated executable and produces both profiling results and a core occupancy map, the latter illustrated below. The illustration shows how the cores were utilized at each time step of execution. In this example, the first core was used for the fast rates (Task1 and Task2) and the second core was used for the slower rates (Task3 and Task4).

You may go back to the Concurrent Execution dialog to try out different core assignments. In particular, you may try to specify no core affinity by leaving the 'Affinity' property as '[]'. This is the default setting which allows Simulink™ to generate multi-threaded code suitable for dynamic scheduling on your multicore processor.
Implement an FFT on a Multicore Processor and an FPGA

This example shows you how to take advantage of a multicore processor target with FPGA acceleration by graphically partitioning a model. This example requires Simulink Coder™ to generate multi-threaded code and HDL Coder™ to generate HDL code. You cannot generate HDL code on Macintosh systems.

Introduction

Several modern processors include multicore processors integrated with FPGA components to create high-performance applications. These require multicore and FPGA programming, including programming of parallel threads, HDL, and communication interfaces between the cores of the system. Simulink™ allows you to take advantage of these approaches by graphically partitioning your algorithms and by assigning the software generated from those partitions to threads on your processor and to modules on your FPGA. The example uses one application level model to generate one executable, consisting of multiple threads and HDL code, to take advantage of the hardware parallelism of the FPGA. This is illustrated in the figure below.
Example Model

Load the example model:

slexMulticoreFPGAExample
**Architecture Definition**

All concurrent execution settings for this model can be accessed in the Concurrent Execution dialog box (Configuration Parameters>Solvers>Configure Tasks).

The first step to implement our algorithm is to define structural elements of our target architecture. This includes structural elements of the hardware, such as the number and type of processing nodes (CPU, FPGA) and the communication channels (AXI, PCI).

This also includes software settings in the model's Configuration Parameters (e.g. System Target File, hardware Implementation, data transfer settings). In this example we have selected the pre-configured target architecture 'Sample architecture'. This architecture uses your desktop as a stand-in for the deployment process.
Partitioning and Mapping the Model

Partition the model to decide which functions run sequentially and which run concurrently.

The example model is partitioned explicitly, consisting of MATLAB System blocks, a Model block, an Atomic Subsystem and an Outport block. Explicit partitioning creates partitions based on these blocks at the root-level of the model. Implicit partitioning, on the other hand, creates partitions based on the block sample times and other scheduling constraints.

After partitioning the model, you can map partitions to CPU tasks and FPGA nodes. You can change the mapping for design space exploration, through the GUI or the API. During mapping, signals will be auto-mapped to channels.

You can change how the blocks are mapped to the threads and to the FPGA in the Concurrent Execution dialog box. You can also find more information in the documentation.
Generate Multithreaded and HDL Code

Double-click on the 'Generate Code and Profile Report' button to generate multithreaded code. In this example, the host computer replaces the target environment. C code is generated for blocks mapped to processor tasks. This code is organized using threads native to the desktop machine. The code snippet shows how threads are created. In addition, HDL code is generated for blocks mapped to hardware nodes. The code snippet below illustrates how the module/entity is created in VHDL.
int main(int argc, char *argv[]) {
    /* Create periodic threads */
    for (i = 0; i < 2; i++) {
        periodicThread[i].index = (int_T) i;
        periodicThread[i].exitFlag = 0;
    }

    /* Create the events that will be used by the thread */
    periodicThread[i].startEvent = CreateEvent(NULL, false, false, NULL);
    periodicThread[i].stopEvent = CreateEvent(NULL, false, true, NULL);

    /* Create the thread in suspended mode */
    periodicThread[i].threadHandle = (HANDLE)_beginthreadex(NULL, 0, 4periodicTask, 4periodicThread[i], CREATE_SUSPENDED, 4periodicThread[i].threadID);
}

ARCHITECTURE rtl OF slexMulticoreInputMdRef IS

   -- Component Declarations
   COMPONENT Input
      PORT( x : IN;
           y1, y2, y3, y4 : OUT);
   END COMPONENT;

   -- Component Configuration Statements
   FOR ALL : Input
      USE ENTITY work.Input(rtl);

   -- Signals
   SIGNAL x : vector_of_std_logic_vector32(0 TO 1023); -- ufix32 [1024]
   SIGNAL Mux_out1 : vector_of_std_logic_vector32(0 TO 511); -- ufix32 [512]
   SIGNAL Mux1_out1 : vector_of_std_logic_vector32(0 TO 511); -- ufix32 [512]

   close_system('slexMulticoreFPGAExample',0);
Multicore Programming of a Field-Oriented Control on Zynq

This example demonstrates how to implement a control algorithm containing multiple rates on Zynq. To take advantage of both the cores and the FPGA hardware, the example uses graphical partitioning approach such that code from different partitions are distributed across the cores and the hardware.

Introduction

In this example, we show a workflow for generating code for a motor control algorithm, and testing the generated code on a Xilinx® Zynq™-7000 SoC ZC702 evaluation board. Motor control algorithm in the example is a Field-Oriented Control algorithm composed of a speed controller (fast component) and a torque controller (slow component). One typical workflow is to generate code for these two components of the controller, upload the generated code to an evaluation board and connect the evaluation board to a real-world motor. The purpose of this example is to show how to follow this workflow. This is illustrated in the block diagram below
In order for this example to be self-contained and since there is redundant computer power on the target processor, we model the motor using one of the tasks of the CPU of the evaluation board.
The example assumes that a Xilinx® Zynq™-7000 SoC ZC702 evaluation board is connected to your computer. You can find the connection and installation steps in the documentation.

This example requires Embedded Coder™ to generate multi-threaded code, HDL Coder (TM) to generate HDL code and Simscape Power Systems (TM) to model the permanent magnet synchronous machine in the example. You cannot generate HDL code on Macintosh systems.
Example Model

The example model consists of two Simulink areas. In the Permanent Magnet Synchronous Machine area, there are two blocks: "Speed and Torque Inputs" block which provides the reference inputs to the feedback system, and "Invertor and Motor" block which is the plant we aim to control in this example. The "Invertor and Motor" block also contains peripherals: a scope that can be used for investigating the simulation results and a UDP sender. The generated code for the UDP sender is responsible for sending the simulation data from the Zynq ZC702 evaluation board to the host machine. The Field-Oriented Control contains the controller blocks "Speed controller" and "Torque controller". Load the model by typing

slexFocZynqExample
**Architecture Definition**

The target architecture in the example is Xilinx Zynq ZC702 evaluation kit. This can be verified by accessing the "Concurrent Execution" dialog box following the steps:

- Go to Simulation>Model Configuration Parameters,
- Click "Configure Tasks" at the Solver pane, under the Additional Options section.

The evaluation board has an ARM Cortex-A9 CPU and a field-programmable gate array (FPGA). There are two tasks running on the ARM CPU. "Torque controller" is mapped to the first task and the blocks that constitute the plant ("Speed and Torque Inputs" block and "Invertor and Motor") are mapped to the second task. "Speed controller" operating at high frequency in the control loop is mapped to the FPGA. These settings can be changed in the "Tasks and Mapping" section of the "Concurrent Execution" dialog box.
Generate Multi-threaded and HDL Code

Press Ctrl-B or click on "Deploy to Hardware" to generate the multi-threaded and HDL code. The generated executable and the FPGA bitstream will be uploaded to Zynq board automatically. In order to perform this step, Zynq board should be connected to the computer and the environment should be set properly. You can consult to the documentation for additional information.

Receive Data from the Zynq

First, open the scope by double-clicking "View Simulation Results", and run the simulation (Simulation>Run) to obtain the following output:
Compare this simulation output with the output of the executable that runs on the Zynq ZC702 board. In order to make this comparison, access the UDP receiver model by double-clicking on the "View Deployment Results". When the UDP receiver model is selected, run the simulation (Simulation>Run). The run command will send a signal to the Zynq ZC702 board to start running the executable on the board. The simulation data will be sent via UDP from the Zynq ZC702 board to the host machine. The captured UDP signal is displayed on the scope of the UDP Receiver model.
Close the Model

close_system('slexFocZynqExample', 0);
Multicore Deployment of a Plant Model

This example illustrates how to take advantage of executing multithreaded code on a multicore processor using graphical partitioning. This example requires Simulink® Coder™ to generate multithreaded code.

Overview

One objective of model-based design is to create realistic models of physical systems and to simulate these models in real-time, for example, to verify controllers using hardware-in-the-loop (HIL). However, as more features are added to the plant model, the computational requirements may exceed the resources available by single-core processing systems.

Partitioning the plant and controller into separate pieces is one way to address the computational needs of complex models. With Simulink® you can partition a plant using Model blocks and then assign the code generated by each sub-model to threads for real-time execution on an HIL system such as Simulink Real-Time™. To see how this works, let us use our host computer as a stand-in for the real-time execution environment and generate real-time multi-threaded code for the following model.

slexMulticoreSolverExample
Symmetric Multicore Processing

The illustration above shows that the code generated for the model is separated into two threads. In this example, the target is assumed to be a Symmetric Multicore Processor, so that the threads are not associated with any particular core. The operating system is responsible for making the best use of the cores when scheduling thread execution. Ideally, to provide maximum flexibility, the number of threads (\(N_t\)) should be greater than the number of cores (\(N_c\)). Double-click on the ‘Generate Code and Profile Report’ button to generate multi-threaded code, profile its execution and visualize the results. The visualization shows a core occupancy map of how the cores were utilized at each time step of execution. It can be seen that the threads float across cores as deemed best by the operating system scheduler. This kind of scheduling is good when the operating system also needs to run other processes.

\[
\begin{align*}
    x_1'(t) &= -x_2(t), \\
    x_2'(t) &= x_1(t),
\end{align*}
\]

where \(x_1(0) = 1\) and \(x_2(0) = 0\).
### Thread Synchronization

Simulink® Coder™ generates code such that the two threads may execute simultaneously and possibly on two different cores. This means that the signal values for $x_1$ and $x_2$ must be synchronized between the two threads. Simulink® provides several options to handle this requirement as illustrated here:

#### Data Transfer Options

<table>
<thead>
<tr>
<th>Defaults</th>
<th>Ensure deterministic transfer (maximum delay)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Periodic signals:</td>
<td>Ensure deterministic transfer (maximum delay)</td>
</tr>
<tr>
<td>Continuous signals:</td>
<td>Ensure data integrity only</td>
</tr>
<tr>
<td>Extrapolation method:</td>
<td>Ensure deterministic transfer (maximum delay)</td>
</tr>
<tr>
<td>Automatically handle</td>
<td>Ensure deterministic transfer (minimum delay)</td>
</tr>
</tbody>
</table>

---

<table>
<thead>
<tr>
<th>Clock Tick / Task</th>
<th>0</th>
<th>10</th>
<th>19</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Thread1</strong></td>
<td>14</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>14</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>14</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td><strong>Thread2</strong></td>
<td>10</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>14</td>
<td>2</td>
<td>14</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>14</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>14</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>14</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>
Using the script below, we will simulate and demonstrate the effect of the deterministic 
modes to understand how Simulink® handles synchronization.

- Reference solution (ode3) - Simulink® is configured to provide the reference solution 
  by synchronizing data at each major and minor time step.
- Zero Order Hold - Each thread is solving a sub-system of equations using its own 
  solver, while synchronizing data only on major time steps.
- Linear extrapolation - In addition to the Zero Order Hold mode, each solver 
  extrapolates data using linear predictions to compensate for data latency errors.

For most systems where the synchronization points are smooth, the linear extrapolation 
mode provides a good tradeoff between communication bottlenecks and numerical 
accuracy.

```matlab
h = figure;
hVal = ishold;
hold on;
mdl = 'slexMulticoreSolverExample';
dt = get_param(mdl, 'DataTransfer');

modes = { ...
    'Ensure deterministic transfer (minimum delay)', ...}
'None', 'k:', ...
'Ensure deterministic transfer (maximum delay)', ...
'Zero Order Hold', 'm', ...
'Ensure deterministic transfer (maximum delay)', ...
'Linear', 'b' ...

};

for i=1:3:length(modes)
dt.DefaultTransitionBetweenContTasks = modes{i};
dt.DefaultExtrapolationMethodBetweenContTasks = modes{i+1};
out = sim(mdl);
plot(out.logsout.get('x1').Values.Time, ...  
    out.logsout.get('x1').Values.Data, ...  
    modes{i+2});
end

legend('Reference solution (ode3)', ...  
    'Zero Order Hold Extrapolation', ...  
    'Linear Extrapolation');
Closing the Models

close_system('slexMulticoreSolverExample',0);
close_system('slexMulticoreSolverMdlref',0);
if ~hVal, hold off; end
delete(h);
Modeling Objects with Identical Dynamics Using For Each Subsystem

This example shows how to model multiple objects with identical dynamics using the For Each Subsystem. The number of objects is parameterized by the length of the input signal.

This example illustrates how you can implement the model `sldemo_metro_basic`, using one For Each Subsystem to model three identical metronomes and another to model the identical effect that the motion of each metronome has on the moving base.

The continuous dynamics of the three identical metronomes in `sldemo_metro_basic` are modeled in identical subsystems: `sldemo_metro_basic/Metronome1`, `sldemo_metro_basic/Metronome2`, and `sldemo_metro_basic/Metronome3`. You can represent these three subsystems as one For Each subsystem. The initial position of the metronomes is passed in as a mask parameter to the For Each subsystem, which is used as an initial condition for one of the integrators. The For Each subsystem is configured to partition this parameter along the first dimension. As a result, an input vector with three elements defines initial positions for three metronomes.
Exploring the Solver Jacobian Structure of a Model
Base model for sldemo_metro_foreach

Figure 1: Metronomes modeled using three subsystems with identical dynamics
On examining the dynamics of the Moving base, note that it involves three identical second-order differential terms corresponding to each metronome. You can replace the three copies with one For Each Subsystem. Instead of nine input signals, the For Each subsystem requires only three input signals: the angular displacements, the angular velocities, and the angular accelerations for all metronomes. The For Each subsystem is configured to partition the input signal along the first dimension and operates on one element from each signal for one iteration (i.e. one metronome).
After implementing the original model with two For Each subsystems, the model is now parameterized for the number of metronomes. To increase the number of metronomes, introduce a new value to the mask parameter.

For example:

\[ \theta_4 = 0.7568; \]
Double-click on the Metronome_i block to open the mask dialog and change Input Positions (Partitioned) to: [\(\Theta_1\) \(\Theta_2\) \(\Theta_3\) \(\Theta_4\)]

Execute the model to see the displacement angles of the four metronomes and the position of the moving base.

```matlab
sim('sldemo.Metro.foreach');
```

**Figure 5:** Simulation animation with For Each subsystem parameterized for four metronomes
Figure 6: Simulation result with For Each subsystem parameterized for four metronomes
Vectorizing a Scalar Algorithm with For Each Subsystem

This example shows how to use the For Each Subsystem. In this example the operations are performed on a vector for simplicity.

Open the example model `sldemo_foreach_vectorized`. This model contains a For Each Subsystem that processes the input signals one by one.

The inputs to the For Each Subsystem are: the signals to process, the coefficients of the FIR filter to use with each of these signals, and the common gain.

Each signal is scalar and needs to be processed individually. Consequently, the For Each Block that controls the partition dimension is set to partition the input signal along dimension 1, by slicing through a partition width of 1. For each input signal, a corresponding set of coefficients must also be partitioned using the same criterion. The gain is common to all the signals, thus this input is not partitioned.

Since the output signal dimension is expected to match the input signal dimension, the concatenation dimension is set equal to the partition dimension. If you prefer to change the signal dimension (a transpose in this case), you can choose 2 as the concatenation dimension.

For the For Each Subsystem, you must set the partition dimension and width, but not the input signal size. A check then ensures that this size is a multiple of the partition width. If no error is detected, the subsystem then calculates the number of independent executions that the enclosed subsystem will perform. These executions are independent, in that, the state associated with a given contained block has a unique value in each of the respective executions.

To see a non-vectorized implementation for this model, double-click on the subsystem in the left corner of the model. This implementation does not use the For Each Subsystem. It mimics its functionality by replicating the subsystem as many times as necessary and also by selecting and concatenating the signals appropriately. This replication process is prone to error and is not scalable- changing the input signal size would require changing the model accordingly.
For Each Subsystem: Vectorizing a Scalar Algorithm

Copyright 2009-2015 The MathWorks, Inc.
Tiled Processing of 2D Signals with For Each Subsystem

This example shows how to use the For Each Subsystem. In this example the operations are performed on matrices.

Open the example model sldemo_foreach_tiled. This model contains a For Each Subsystem that processes submatrices of a matrix input signal.

To process submatrices of a matrix signal, you will need to use nested For Each Subsystems. The outer subsystem can be used to partition the signal along the first dimension, while the inner one can be used to partition along the second dimension. This order is arbitrary, and can be changed without affecting the result.

In this example a [4x4] matrix is to be partitioned into [2x2] submatrices. The outer subsystem partitions the signal along the first dimension. The resulting subarrays are then partitioned by the inner subsystem along the second dimension. Concatenation of the outputs respects the partitioning order to preserve the shape of the input signal. Note, however, that the output dimensions change since the processing of the submatrices results in scalar outputs.

To see a tile repetition implementation for this model, double-click on the subsystem in the left corner of the model window. This implementation does not use the For Each Subsystem. Instead, it mimics the functionality of the For Each Subsystem by replicating the subsystem as many times as necessary and by selecting and concatenating the signals appropriately. This replication process is prone to error and is not scalable- changing the input signal size would require changing the model accordingly.
For Each Subsystem:
Tiled Processing of 2-D Signals

Tile processing:
slice across first dimension

Copyright 2009-2015 The MathWorks, Inc.
Neighborhood Processing Using For Each Subsystems

This example shows how to use the For Each Subsystem block to process submatrices data when neighboring submatrices overlap each other. To demonstrate this data processing approach, the example implements the image edge detection with linear filters using the For Each Subsystem block. Each For Each Subsystem block has a For Each block that needs to be configured for partitioning and concatenation.

Open the example model `slexForEachOverlapExample`. This model uses a 3-by-3 kernel to implement the edge detection application for a given two-dimensional image. Because of the two-dimensional input signal, this example uses nested for each subsystems.

- The outer for each subsystem partitions the signal along the first dimension. The outer For Each block is configured to partition the input signal along dimension 1. The block specifies a partition width of 3 and a partition offset of -2. (A partition offset of -2 indicates that each slice overlaps its neighboring slices by 2 elements.)

- The inner for each subsystem partitions the signal along the second dimension. The inner For Each block has the same settings as the outer For Each block except that the inner block is set to partition and concatenate signals along dimension 2.

The partitioning order is arbitrary. To ensure correctness of the result, the concatenation order of the outputs follows the partitioning order.

For simplicity, this example does not handle data padding. As a result, the size of the output image is slightly smaller than the size of the input image.
The result of the comparison between the original image and the processed image is shown below.
**Figure 1:** Original image vs. processed image
Using a Project with SVN

This example shows how to use a project to manage the files within your design. Starting with an existing project that already uses source control, this example shows how to find and manage the files within your project. A common workflow illustrated by this example is fulfilling a change request for your design.

**Setting Up the Example Files and Opening the Project**

Run the following commands to create and open a working copy of the project files.

```
Simulink.ModelManagement.Project.projectDemo('airframe', 'svn');
rebuild_s_functions('no_progress_dialog');
```

Building with 'Microsoft Visual C++ 2017 (C)'.
MEX completed successfully.

The project example copies files to a new folder so that you can edit them and use them under Subversion (SVN) version control.

This project is configured to manage its folders on the MATLAB search path and some working folders.
Managing the Project Path

The project defines a project path. Folders specified in the project path are added to the MATLAB search path at project start up and removed from the MATLAB search path when the project closes.

To edit or view this example's project path click the "Project Path" button in the Project toolstrip tab.

Managing Working Folders

You can use the project to define the following working folders: the simulation cache folder, the code generation folder and the current directory when you open the project.

Click the "Details" button in the Project toolstrip tab to change any of these working folders.
**Startup and Shutdown**

Project files, such as MATLAB scripts, can be set to run when the project opens or closes.

To specify which project files are run during startup and shut down, click the "Startup Shutdown" button in the project toolstrip tab.

On the Manage Project Startup and Shutdown dialog box, you can specify to start Simulink on opening the project. You can also choose to refresh customizations (run sl_customization files) on project startup and shutdown.

**Project Shortcuts**

Project shortcuts are shown in the "Project Shortcuts" toolstrip tab.

This example project has a shortcut for: the top-level mode, a utility function to rebuild s-functions and a design description document.

Click the shortcut "F14 Model" on the Project Shortcuts toolstrip tab to open the root model for this project.

Click the shortcut "Rebuild Project's S-functions" on the Project Shortcuts toolstrip tab to generate the S-Function.

To create new shortcut for a project file click the "New Shortcut" button on the Project Shortcuts toolstrip tab.

To edit a shortcut right click on a shortcut in the Project Shortcuts toolstrip tab and select "Edit Shortcut".

**Using Dependency Analysis**

Run a file dependency analysis on the files within your project to check that all the files that are required by the project are stored with in it.

Click the Dependency Analysis node in the Views pane.

On the Dependency Analysis tab, click the Analyze button.

Review files reported in the Dependencies node. If problems are detected then all other results are filtered out. To see all the results of the file dependency analysis, click on the warning icon in the toolbar to turn off the "show only problem files" filter.
You can see from the results of the file dependency analysis that the S-Function binary, timesthree, is required by the project but is not currently part of it.

Click on timesthree in the Dependencies table to view where it is used within this project. In this example, it is used by f14_airframe.slx.

You may want to add binary files to your project or, as in this project, provide a utility script that regenerates them from the source code that is part of the project.

Right-click timesthree and select "Add External File". The next time you run dependency analysis, this file will not be marked as a problem file.

**Modifying Files**

You can open files for editing from the project by double-clicking, or by right-clicking and selecting "Open".

Try opening and making changes to one of the utility MATLAB files, or one of the Simulink models.

**Using The Modified Files View**

Click on Modified filter in the Files view to see the files that you have modified in your working copy.

You can use the comparison tool from the project to understand the changes you have made, perhaps as part of a peer-review process.

Right-click a file in the Modified File view and select "Compare to Ancestor".

This launches an appropriate comparison using the MATLAB Comparison Tool, comparing the modified version of the file in your working copy against its ancestor stored in your version control tool. If you select a Simulink model this runs a Simulink® Model Comparison.

Click the commit button in the Project toolstrip tab to commit your changes to Source Control.

The files stored in the resources/project folder are internal project definition files generated by your changes. The project definition files allow you to add metadata to files, for example, by creating shortcuts, adding labels, and adding a project description. Project definition files also define the files that are added to your project. You can review changes in revisions of project definition files like any other project files.
**Viewing Project Information**

Click the "Details" button in the Project toolstrip tab to see information about the currently open project, including a description and the location of the project root folder.

**Viewing Source Control Information**

On the Project tab, the source control section contains buttons to perform source control operations such as commit. To see more information on the tool being used, click the details button in this section.

For source control information on individual files (for example modified, added), see the Git column in the file management views.

**Using Project File Views**

In the Files view, use the Project Files view to manage the files within your project. Only the files that are in your project are shown.

In the Files view, click the Project Files view button and select the All Files view to see all the files in your working copy. This shows all the files that are under the project root, not just the files that are in the project. This view is useful for adding files to the project that exist within your working copy, but which are not yet part of the project.

In any file view, click the List button at the top right to view the files as a list.

Click the "cog" icon Actions button at the top right to customize the views and to sort files.

Right click the heading row of a file view to group files.

**Further Information**

Project documentation
Using a Project with Git

This example shows how to use a project to manage the files within your design. Starting with an existing project that is already checked into source control, this example shows how to find and manage the files within your project. A common workflow illustrated by this example is fulfilling a change request for your design.

Setting Up the Example Files and Opening the Project

Run the following commands to create and open a working copy of the project files.

Simulink.ModelManagement.Project.projectDemo('airframe', 'git');
rebuild_s_functions('no_progress_dialog');

Starting: Simulink
Building with 'Microsoft Visual C++ 2017 (C)'.
MEX completed successfully.

The project example copies files to a new folder so that you can edit them and use them under Git version control.

This project is configured to manage its folders on the MATLAB search path and some working folders.
Managing the Project Path

The project defines a project path. Folders specified in the project path are added to the MATLAB search path at project start up and removed from the MATLAB search path when the project closes.

To edit or view this example's project path click the "Project Path" button in the Project toolstrip tab.

Managing Working Folders

You can use the project to define the following working folders: the simulation cache folder, the code generation folder and the current directory when you open the project.

Click the "Details" button in the Project toolstrip tab to change any of these working folders.
**Startup and Shutdown**

Project files, such as MATLAB scripts, can be set to run when the project opens or closes.

To specify which project files are run during startup and shut down, click the "Startup Shutdown" button in the project toolstrip tab.

On the Manage Project Startup and Shutdown dialog box, you can specify to start Simulink on opening the project. You can also choose to refresh customizations (run sl_customization files) on project startup and shutdown.

**Project Shortcuts**

Project shortcuts are shown in the "Project Shortcuts" toolstrip tab.

This example project has a shortcut for: the top-level mode, a utility function to rebuild s-functions and a design description document.

Click the shortcut "F14 Model" on the Project Shortcuts toolstrip tab to open the root model for this project.

Click the shortcut "Rebuild Project's S-functions" on the Project Shortcuts toolstrip tab to generate the S-Function.

To create new shortcut for a project file click the "New Shortcut" button on the Project Shortcuts toolstrip tab.

To edit a shortcut right click on a shortcut in the Project Shortcuts toolstrip tab and select "Edit Shortcut".

**Using Dependency Analysis**

Run a file dependency analysis on the files within your project to check that all the files that are required by the project are stored with in it.

Click the Dependency Analysis node in the Views pane.

On the Dependency Analysis tab, click the Analyze button.

Review files reported in the Dependencies node. If problems are detected then all other results are filtered out. To see all the results of the file dependency analysis, click on the warning icon in the toolbar to turn off the "show only problem files" filter.
You can see from the results of the file dependency analysis that the S-Function binary, timesthree, is required by the project but is not currently part of it.

Click on timesthree in the Dependencies table to view where it is used within this project. In this example, it is used by f14_airframe.slx.

You may want to add binary files to your project or, as in this project, provide a utility script that regenerates them from the source code that is part of the project.

Right-click timesthree and select "Add External File". The next time you run dependency analysis, this file will not be marked as a problem file.

**Modifying Files**

You can open files for editing from the project by double-clicking, or by right-clicking and selecting "Open".

Try opening and making changes to one of the utility MATLAB files, or one of the Simulink models.

**Using The Modified Files View**

Click on Modified filter in the Files view to see the files that you have modified in your working copy.

You can use the comparison tool from the project to understand the changes you have made, perhaps as part of a peer-review process.

Right-click a file in the Modified File view and select "Compare to Ancestor".

This launches an appropriate comparison using the MATLAB Comparison Tool, comparing the modified version of the file in your working copy against its ancestor stored in your version control tool. If you select a Simulink model this runs a Simulink® Model Comparison.

Click the commit button in the Project toolstrip tab to commit your changes to Source Control.

The files stored in the resources/project folder are internal project definition files generated by your changes. The project definition files allow you to add metadata to files, for example, by creating shortcuts, adding labels, and adding a project description. Project definition files also define the files that are added to your project. You can review changes in revisions of project definition files like any other project files.
Viewing Project Information

Click the "Details" button in the Project toolstrip tab to see information about the currently open project, including a description and the location of the project root folder.

Viewing Source Control Information

On the Project tab, the source control section contains buttons to perform source control operations such as commit. To see more information on the tool being used, click the details button in this section.

For source control information on individual files (for example modified, added), see the Git column in the file management views.

Using Project File Views

In the Files view, use the Project Files view to manage the files within your project. Only the files that are in your project are shown.

In the Files view, click the Project Files view button and select the All Files view to see all the files in your working copy. This shows all the files that are under the project root, not just the files that are in the project. This view is useful for adding files to the project that exist within your working copy, but which are not yet part of the project.

In any file view, click the List button at the top right to view the files as a list.

Click the "cog" icon Actions button at the top right to customize the views and to sort files.

Right click the heading row of a file view to group files.

Further Information

Project documentation
Using a Project

This example shows how to use a project to manage the files within your design. Starting with an existing project that already uses source control, this example shows how to find and manage the files within your project. A common workflow illustrated by this example is fulfilling a change request for your design.

Setting Up the Example Files and Opening the Project

Run the following commands to create and open a working copy of the project files.

```matlab
Simulink.ModelManagement.Project.projectDemo('airframe', 'git');
rebuild_s_functions('no_progress_dialog');
```

Building with 'Microsoft Visual C++ 2017 (C)'.
MEX completed successfully.

The project example copies files to a new folder so that you can edit them and use them under Git version control.

This project is configured to manage its folders on the MATLAB search path and some working folders.
Managing the Project Path

The project defines a project path. Folders specified in the project path are added to the MATLAB search path at project start up and removed from the MATLAB search path when the project closes.

To edit or view this example's project path click the "Project Path" button in the Project toolstrip tab.

Managing Working Folders

You can use the project to define the following working folders: the simulation cache folder, the code generation folder and the current directory when you open the project.

Click the "Details" button in the Project toolstrip tab to change any of these working folders.
Startup and Shutdown

Project files, such as MATLAB scripts, can be set to run when the project opens or closes.

To specify which project files are run during startup and shut down, click the "Startup Shutdown" button in the project toolstrip tab.

On the Manage Project Startup and Shutdown dialog box, you can specify to start Simulink on opening the project. You can also choose to refresh customizations (run sl_customization files) on project startup and shutdown.

Project Shortcuts

Project shortcuts are shown in the "Project Shortcuts" toolstrip tab.

This example project has a shortcut for: the top-level mode, a utility function to rebuild s-functions and a design description document.

Click the shortcut "F14 Model" on the Project Shortcuts toolstrip tab to open the root model for this project.

Click the shortcut "Rebuild Project's S-functions" on the Project Shortcuts toolstrip tab to generate the S-Function.

To create new shortcut for a project file click the "New Shortcut" button on the Project Shortcuts toolstrip tab.

To edit a shortcut right click on a shortcut in the Project Shortcuts toolstrip tab and select "Edit Shortcut".

Using Dependency Analysis

Run a file dependency analysis on the files within your project to check that all the files that are required by the project are stored with in it.

Click the Dependency Analysis node in the Views pane.

On the Dependency Analysis tab, click the Analyze button.

Review files reported in the Dependencies node. If problems are detected then all other results are filtered out. To see all the results of the file dependency analysis, click on the warning icon in the toolbar to turn off the "show only problem files" filter.
You can see from the results of the file dependency analysis that the S-Function binary, timesthree, is required by the project but is not currently part of it.

Click on timesthree in the Dependencies table to view where it is used within this project. In this example, it is used by f14_airframe.slx.

You may want to add binary files to your project or, as in this project, provide a utility script that regenerates them from the source code that is part of the project.

Right-click timesthree and select "Add External File". The next time you run dependency analysis, this file will not be marked as a problem file.

**Modifying Files**

You can open files for editing from the project by double-clicking, or by right-clicking and selecting "Open".

Try opening and making changes to one of the utility MATLAB files, or one of the Simulink models.

**Using The Modified Files View**

Click on Modified filter in the Files view to see the files that you have modified in your working copy.

You can use the comparison tool from the project to understand the changes you have made, perhaps as part of a peer-review process.

Right-click a file in the Modified File view and select "Compare to Ancestor".

This launches an appropriate comparison using the MATLAB Comparison Tool, comparing the modified version of the file in your working copy against its ancestor stored in your version control tool. If you select a Simulink model this runs a Simulink® Model Comparison.

Click the commit button in the Project toolstrip tab to commit your changes to Source Control.

The files stored in the resources/project folder are internal project definition files generated by your changes. The project definition files allow you to add metadata to files, for example, by creating shortcuts, adding labels, and adding a project description. Project definition files also define the files that are added to your project. You can review changes in revisions of project definition files like any other project files.
**Viewing Project Information**

Click the "Details" button in the Project toolstrip tab to see information about the currently open project, including a description and the location of the project root folder.

**Viewing Source Control Information**

On the Project tab, the source control section contains buttons to perform source control operations such as commit. To see more information on the tool being used, click the details button in this section.

For source control information on individual files (for example modified, added), see the Git column in the file management views.

**Using Project File Views**

In the Files view, use the Project Files view to manage the files within your project. Only the files that are in your project are shown.

In the Files view, click the Project Files view button and select the All Files view to see all the files in your working copy. This shows all the files that are under the project root, not just the files that are in the project. This view is useful for adding files to the project that exist within your working copy, but which are not yet part of the project.

In any file view, click the List button at the top right to view the files as a list.

Click the "cog" icon Actions button at the top right to customize the views and to sort files.

Right click the heading row of a file view to group files.

**Further Information**

Project documentation
Perform Impact Analysis with a Project

This example shows how to use the project dependency analysis tools to perform file-level impact analysis. Starting with an existing project that uses Git source control, this example shows how to determine which files are impacted by modifications to the project. This workflow can help you find and run the tests needed to validate a change to your design.

Set Up the Example Files and Open the Project

1. Run the following commands to create and open a working copy of the "sldemo_slproject_airframe" example.

`Simulink.ModelManagement.Project.projectDemo('airframe', 'git');
rebuild_s_functions('no_progress_dialog');`

Building with 'Microsoft Visual C++ 2017 (C)'.
MEX completed successfully.

The project example copies files to a new folder so that you can edit them and use them under Git version control.

Perform Dependency Analysis

Run a dependency analysis on the files in your project.

2. Click the Dependency Analysis node.

3. In the Dependency Analysis view, click Analyze.

The dependency graph displays the structure of your project. Each box in the graph represents a file, and each arrow is a dependency.

Modify the Project

4. Make a modification to `vertical_channel.slx` and save the model. This can be a trivial modification such as moving a block.

The project can determine the source control status of each file. You can display this information in the dependency graph by changing the file highlighting.

5. On the Dependency Analysis tab, click Group By > Git.
The color of each file in the graph now represents its source control status. The status that each color represents is shown in the Git legend in the top left of the view. For vertical_channel.slx the source control status is Modified, because you made a change to the file.

**Find the Impact of the Modification**

You can use the Impact Analysis tools on the Dependency Analysis tab to analyze the graph to find impacted and required files. You choose the files to analyze, and the files to find. The tool filters the graph to show only the files found in the analysis. You can clear the filter using the X button on the filter bar.

6. On the Dependency Analysis tab, click Select > Modified Files, followed by Find > Files Impacted by Selection.

The files displayed in the graph are filtered to show only those impacted by the modifications that you have made to the project.

**Determine Which Tests Need to Be Run**

The labels that are attached to each file can be displayed on the graph by changing the file highlighting. You can use this to display information about each file.

All projects come with a Classification label category that contains some common file classifications. This includes a Test label that is used to identify which files are tests.

7. On the Dependency Analysis tab, click Group By > Classification.

Each file in the graph is now colored by its file Classification. For the files impacted by the change to vertical_channel.slx there are some Design files and a Test file. This identifies the tests that are impacted by the modification. These are the tests that need to be run to validate the design changes.

**Export the Results and Run Tests**

You can send files to other tools using the Export menu on the Dependency Analysis tab. If no files are selected, you export all files in the currently filtered view, otherwise only the selected files are exported. You can click items in the legends to select groups of files to export.

8. Click Test in the Classification legend to select all test files.

The Custom Task dialog appears. The test files you exported from the impact analysis are selected.

10. To run the selected tests, select Run Unit Tests in the drop down menu. Click Run Task.

The Run Unit Tests custom task runs MATLAB unit tests and creates a simple summary report. Look in the Results pane in the Custom Task Report to see the report for each test file run.

- If you want to export the impacted files to the workspace, in the Dependency Analysis view you can select the Save to Workspace export option. This creates a cell array of file paths to the exported files.

- If you want to view the impacted files in the Files view, in the Dependency Analysis view you can select the Show in Files View export option. This changes the view to the Files view and selects the exported files.

Further Information

Project documentation
Referencing Projects from Another Project

This example shows how to use referenced projects as independent components within your project.

Setting Up the Example Files and Opening the Project

1. Run the following commands to create and open a working copy of the project files.

```matlab
Simulink.ModelManagement.Project.projectDemo('airframe_refs', 'svn', true);
```

How Referenced Projects Work

When the parent project is loaded, the referenced projects are loaded first, to add project paths from the referenced project to the MATLAB search path, and to run startup shortcuts from the referenced projects.

- You can run the shortcuts in referenced projects from the parent project, using the Project Shortcuts tab.
- You can examine and open referenced projects by clicking the References node in the Views pane.

Example Referenced Projects

This project provides an airframe simulation which brings in functionality from these other projects by referencing them:

- Signal Multiplier: This referenced project shows how to develop an S-function as an independent project. The referenced project provides source code, build support and path management. The build directory is on the project path, so it is also on the path for the parent project. You can run the "rebuild s-function" shortcut from the parent project on the Project Shortcuts tab.
- Wind Gust library: This referenced project provides a library and a data dictionary that contains the required bus objects for the library. This project's path ensures that the library is set up for use in the parent project.

Further Information

Referenced Projects
Automate Label Management in a Project

This example shows how to use the project functions to manage labels.

**Open the Airframe Example Project**

Run the following commands to create and open a working copy of the "sldemo_slproject_airframe" example.

```matlab
Simulink.ModelManagement.Project.projectDemo('airframe');
rebuild_s_functions('no_progress_dialog');
```

Building with 'Microsoft Visual C++ 2017 (C)'.
MEX completed successfully.

**Get a Project Object.**

Use the `currentProject` function to get a project object to manipulate the currently open project at the command line.

```matlab
project = currentProject;
```

**View the Labels of a File**

Examine the `Files` property of the project. The `Files` property contains an array of file objects, one for each file in the project.

```matlab
files = project.Files;
disp(files);
```

1x30 ProjectFile array with properties:

- Path
- Labels
- Revision
- SourceControlStatus

You can use indexing to access files in this array. The following command gets file number 2. Each file has two properties describing its path and attached labels.

```matlab
aFile = files(2);
disp(aFile);
```

ProjectFile with properties:
Find information about a file's attached labels by indexing into the file object's Labels property. The following command gets the first label attached to this particular file.

```matlab
default
label = aFile.Labels(1);
disp(label);

Label with properties:

    File: "C:\Users\batserve\MATLAB\Projects\examples\airframe5\custom_tasks\analyzeModelFiles.m"
    DataType: 'none'
    Data: []
    Name: "Analysis"
    CategoryName: "Classification"
```

**Attach a Label to a Subset of Files.**

The following code attaches the label 'Design' in the 'Classification' category to all files in the project with the .m file extension.

```matlab
First get the list of files:

files = project.Files;

Then loop through each file and attach the label 'Design' from the 'Classification' category if the file has the extension .m.

```matlab
for fileIdx = 1:numel(files)
    file = files(fileIdx);
    [~, ~, fileExtension] = fileparts(file.Path);
    if strcmp(fileExtension,'.m')
        addLabel(file, 'Classification', 'Design');
    end
end
```
Find a Named Label

You can set and query data on a label that is attached to a file. To do this, you first need to find the file object. You can do this by looping through all files in the project, as shown in the previous step. Alternatively, you can use the findFile function on the project.

The following code finds the file object for the file 'utilities/rebuild_s_functions.m'.

```matlab
pathToLocate = fullfile('utilities','rebuild_s_functions.m');
file = findFile(project, pathToLocate);
```

Examine the Labels property to get an array of Label objects, one for each label attached to the file.

```matlab
labels = file.Labels;
disp(labels);
```

Label with properties:
- **File**: "C:\Users\batserve\MATLAB\Projects\examples\airframe5\utilities\rebuild_s_functions.m"
- **DataType**: 'none'
- **Data**: []
- **Name**: "Design"
- **CategoryName**: "Classification"

To find a label by name, use findLabel on the file object.

```matlab
label = findLabel(file, 'Classification','Design');
disp(label);
```

Label with properties:
- **File**: "C:\Users\batserve\MATLAB\Projects\examples\airframe5\utilities\rebuild_s_functions.m"
- **DataType**: 'none'
- **Data**: []
- **Name**: "Design"
- **CategoryName**: "Classification"

Create a New Category

You must create new labels before you can attach them to a file. You define labels in categories, giving each category a name and supported data type.
The following code creates a category of labels called Engineers which can be used to denote file ownership in a project. These labels have the char datatype for attaching String data.

```matlab
createCategory(project, 'Engineers', 'char');
engineersCategory = findCategory(project, 'Engineers');
createLabel(engineersCategory, 'Sam');
createLabel(engineersCategory, 'Pat');
createLabel(engineersCategory, 'Alex');
```

You can now attach an Sam label from the 'Engineers' category to a file in the project.

```matlab
addLabel(file, 'Engineers', 'Sam');
label = findLabel(file, 'Engineers', 'Sam');
```

**Set Label Data**

The following command sets the data for the attached label.

```matlab
label.Data = 'Maintenance responsibility';
disp(label)
```

```
Label with properties:

    File: "C:\Users\batserve\MATLAB\Projects\examples\airframe5\utilities\rebu..."
    DataType: 'char'
    Data: 'Maintenance responsibility'
    Name: "Sam"
    CategoryName: "Engineers"
```

**Further Information**

Project documentation
Running Custom Tasks with a Project

This example shows how to apply a custom task to a set of files managed by project. The example custom task analyzes the Simulink models which are in the project, reporting the number of blocks in each model.

Setting Up the Example Files and Opening Project

1. Run the following commands to create and open a working copy of the "sldemo_slproject_airframe" example.

   ```matlab
   Simulink.ModelManagement.Project.projectDemo('airframe');
   rebuild_s_functions('no_progress_dialog');
   
   Building with 'Microsoft Visual C++ 2017 (C)'.
   MEX completed successfully.
   
   The project example copies files to a new folder so that you can edit them and put them under local version control.
   
Select a Custom Task

2. Click the Custom Task button in the Tools section of the project toolstrip tab.

   You define a custom task with a MATLAB® function. The example Airframe project contains example custom tasks in the custom_tasks folder.

3. The "Custom Task" menu lists available custom tasks.

   To view, edit, and create custom tasks, click Custom Tasks > Manage Custom Tasks in the Tools section of the Project toolstrip tab.

4. Select the 'Analyze Model Files' custom task.

   The function name of your selected custom task appears in the "Custom task" edit box. The example analyzeModelFiles adds a label from the category "Metrics" to each model file in the project. Labels in this category have numerical data. The custom task will count the number of blocks in each model and attach this number to the label.

Select Files to Include in the Custom Task

5. In the Custom Task dialog box, verify that all model files have selected check boxes in the Include column.
Running a Custom Task

6. Click the Run Task button on the bottom right of the Custom Task dialog box.

The results for a selected file are also shown in the Results pane at the bottom of the dialog box. This can be useful when the results returned are long, or contain HTML markup.

The following example shows the dialog box after running the custom task on some models. You can customize the columns to show with the "cog" icon Actions button at the top right.
**Editing an Existing Custom Task**

Custom Tasks are MATLAB functions. Edit your custom task with the MATLAB editor. In the following steps, you modify the custom task to use the project API to add a label with data, as well as saving any dirty model files.

7. Run the following MATLAB code to create a Metrics category and a Block Count label in the project.

```matlab
project = currentProject;
category = createCategory(project, 'Metrics', 'double');
```

8. Double-click analyzeModelFiles.m (or right-click and select Open) to edit it in the MATLAB editor.

9. Add the following lines just after the sprintf command:

```matlab
[~, compileStats] = sldiagnostics(name, 'CompileStats');
addLabel(projectFile, 'Metrics', 'CPU Compile Time', sum([compileStats.Statistics.CPUTime]));
```

You can use the MATLAB editor to set breakpoints and debug a custom task function, just as with any other MATLAB function.

If you rerun the custom task, it adds the CPU Compile Time label to each model file that can be compiled, and attaches data to the label showing the total time for all compilation phases for the model. Models that cannot be compiled show "Failed to analyze file" in the Custom Task Report, and details display as a warning in the command window. Examine the custom task analyzeModelFiles.m to see how to handle errors.

To view the new metrics data, either show the Metrics column in the Custom Task Report, or look in the project files view.

**Creating a New Custom Task**

Create a new custom task by creating a new MATLAB function. Your custom tasks must:

- Be saved on the MATLAB path.
- Accept a single input argument: a full path to a file.
- Return a single output argument.

To create custom tasks, click Custom Tasks > Manage Custom Tasks in the Tools section of the project toolstrip tab. In the Manage Custom Tasks dialog box, click Add to open a
new file with instructions that guide you to create a custom task with the correct function signature.

10. Click "Add" and select the "Add Using New Script" menu item.

A file dialog opens asking you to choose where to create the new custom task. The custom task must be saved on the MATLAB path to run.

11. Provide a file name and save the file in the 'custom_tasks' folder within the project.

The MATLAB editor opens the file pre-populated with a simple example custom task.

12. To create your new custom task, edit the contents of the example custom task function and save.

**Further Information**

Project documentation
Upgrade Simulink Models Using a Project

Easily upgrade all the models in your project using the Upgrade Project tool in a project.

Setting Up the Example Files and Opening the Project

Run the following commands to create and open a working copy of the project files.

```matlab
Simulink.ModelManagement.Project.projectDemo('airframe', 'git');
rebuild_s_functions('no_progress_dialog');
```

Building with 'Microsoft Visual C++ 2017 (C)'.
MEX completed successfully.

The project example copies files to a new folder so that you can edit them and use them under Git version control. Placing your files under version control lets you compare the upgraded model files with the original versions.

Automatically Upgrade All Project Models and MATLAB Code at Once

Upgrade all models and MATLAB code in your project to the latest release using a simple workflow. The Upgrade Project tool can apply fixes automatically when possible, upgrade all model hierarchies in the project at once, and produce a report.

In the Project tab click Run Checks > Upgrade.
This loads the upgrade advisor checks and runs a dependency analysis on your project to ensure the project hierarchy is handled correctly.

Click Upgrade to run all upgrade checks on all models, libraries and MATLAB code in your project. If any check fails and there is a fix available, the project automatically applies the fix and runs the relevant checks again to ensure they now pass.

Inspect the results of the upgrade in the report.
Further Information

Project documentation  Simulink Upgrade Advisor documentation  Upgrade All Project Models documentation
Convert from MDL to SLX in a Project and Preserve Revision History

Project Tools for Migrating Model Files to SLX

A project helps you upgrade model files from MDL format to SLX format. The default file format for Simulink® models in R2012b and subsequent releases is SLX. You can use the project integrity checks to automatically add the new SLX file to your project, remove the MDL file from the project, and preserve the revision history of your MDL file with the new SLX file. You can then commit your changes to source control and maintain the continuity of your model file history.

Open the Airframe Example Project

Create and open a working copy of the airframe project using Git™ source control.

Simulink.ModelManagement.Project.projectDemo('airframeConvertToSlx', 'git');
rebuild_s_functions('no_progress_dialog');

Building with 'Microsoft Visual C++ 2017 (C)'.
MEX completed successfully.

Find All MDL Files in the Project and Convert to SLX

In the models folder, check if this project contains models saved in MDL format.

To find all the MDL files in this project and resave them as SLX files, use the project API:

```matlab
project = currentProject;
allProjectFiles = project.Files;
for fileIndex = 1:numel(allProjectFiles)
    projectFile = allProjectFiles(fileIndex);
    [folder, name, ext] = fileparts(projectFile.Path);
    if strcmp(ext, '.mdl')
        % This is a MDL file. Load it and then save as an SLX file.
        mdlModel = fullfile(folder, [name + ext]);
        slxModel = fullfile(folder, [name + '.slx']);
        load_system(mdlModel);
        % Save a new copy of this file in SLX format:
        save_system(name, slxModel);
        close_system(name,0);
    end
end
```
To see the results, in the **Files** view, change the view from **Project (number of files)** to **All**. Expand the models folder. Simulink® saves the model in SLX format and creates a backup file by renaming the MDL file to `filename.mdl.releasename`, for example, `AnalogControl.mdl.r2019b`. The project also reports the original name of the MDL file as missing.

**Run the Project Checks**

To resolve these issues, on the Project tab, click **Run Checks > Check Project** to run the project integrity checks. The checks look for MDL files converted to SLX and offer automatic fixes if that check fails. Click the **Fix** button to view recommended actions and decide whether to make the changes.
The Missing Files dialog box offers to remove the missing MDL file from the project and add the new SLX file to the project.
Project checks rerun after you click **Yes** to perform the fix. Close the Project Integrity Checks dialog box.
Verify Changes After Upgrade to SLX

Click the Modified files view of the project. Project checks added the newly created SLX files to the project and scheduled the original MDL files for removal.
Commit Changes and Verify Revision History

On the Project tab, in the Source Control section, click **Commit**. Enter a comment for your submission in the dialog box, for example, Convert to SLX, and click **Submit**. Right-click the new SLX model. Select **Source Control > Show Revisions**. In the File Revisions dialog box, verify that the previous revision is filename.mdl, for example, AnalogControl.mdl. The revision history of the previous model file is preserved with the new SLX file.

Further Information

Project documentation
Export a Subset of a Project Using an Export Profile

This example shows how to export a release version of a project using an export profile. The example export profile streamlines the exported project by removing files from the project which are only useful for the development of the project, but are not necessary for running the project's top-level models.

Set Up the Example Files and Open the Project

1. Run the following commands to create and open a working copy of the airframe project.

```matlab
Simulink.ModelManagement.Project.projectDemo('airframe');
rebuild_s_functions('no_progress_dialog');
```

Building with 'Microsoft Visual C++ 2017 (C)'.
MEX completed successfully.

The classification label set is used to define the purpose of each file in the project. This example project adds several new labels to the 'classification' label set: 'analysis', 'upgrade', and 'utility'. These labels are used to identify files which are used for custom analysis tasks, upgrading to the latest version and general purpose utility functions.

Some files in the project are useful for the development of the project, but do not need including in a release version of the project when it is exported. For this example, files labelled in the project with the labels 'analysis', 'test', 'upgrade' and 'artifact' do not need including in the release version of the project.

This example demonstrates how the export profile management feature can be used to exclude files from a project export based on the attached labels.

Open the Export Profile Manager Tool

To view the available export profiles for a project:

2. Click the "Share" button in the Project toolstrip tab.

3. Select the menu item "Manage Export Profiles..."

The Manage Export Profiles dialog box will appear.

The left hand panel shows all of the defined export profiles. The airframe project defines one export profile called 'Release', which should be already be selected in the dialog box.
The export profile shows two panels. The first panels shows the labels which will cause a file to not be exported if attached to a file. The release profile has been defined such that any file which has the 'analysis', 'test', 'artifact', or 'upgrade' label attached will not be exported.

The second panel defines labels which will not be included in the exported project. Here the labels 'analysis' and 'upgrade' are removed because no files in the exported project will use them.

The export profile interface allows new profiles to be defined and existing profiles to be removed.

**Using an Export Profile When Exporting a Project**

To use an export profile:

4. Click the "Share" button in the Project toolstrip tab.

5. Select the "Zip Archive".

6. In the project archive dialog, select the export profile to be applied. Select the 'Release' profile to use the profile that is defined by the airframe project.

7. Click the Archive button to export the project using the selected export profile.

**Further Information**

Project documentation
Creating Projects Programmatically

This example shows how to use the project API to create and set up a new project. It shows how to create a project from the command line, add files and folders, set up the project path, define project shortcuts and create a reference to the new project in another project.

Setting Up the Example Files

Run the following commands to create a working copy of the project files.

```matlab
Simulink.ModelManagement.Project.projectDemo('airframe_api', [], true);
```

Create the Wind Gust Project

The following commands will create the wind gust project. This project contains a library and a data dictionary which will be used by the top level project through a project reference.

Create the project and set the project name.

```matlab
mainProject = currentProject;
windGustFolder = fullfile(mainProject.RootFolder, '..', 'refs', 'Wind Gust Library');
windGust = matlab.project.createProject(windGustFolder);
windGust.Name = 'Wind Gust Library';
```

% Add the files and folders to the new project.

```matlab
addFolderIncludingChildFiles(windGust, 'data');
addFile(windGust, 'wind_gust_lib.slx');
```

% Add folders to the project path

```matlab
addPath(windGust, windGust.RootFolder);
addPath(windGust, 'data');
```

% Define the project shortcuts.

```matlab
shortcut = addShortcut(windGust, 'wind_gust_lib.slx');
shortcut.Group = 'Top Level Model';
```
**Add a Project Reference**

The following commands will add the new Wind Gust Library project to the top level project as a project reference.

```matlab
reload(mainProject);
addReference(mainProject, windGust);
```

**Further Information**

Referenced Projects
Compare and Merge Simulink Models

This example shows how to view and merge the changes between two Simulink® models from the comparison report.

This example shows you how to use the comparison report in MATLAB to understand what has changed between two different designs. An engineer has made some changes to the model slxml_aircraft1 and the resulting model is slxml_aircraft2. The Simulink Model Comparison report will show you what these changes are and enable you to create another model containing changes from each of the two source models.

**Compare The Models**

Setup the example files by running the following command.

```matlab
slxmlcomp.internal.examples.setupCompareMergeExample();
% Then use visdiff to compare the models.
visdiff('slxml_aircraft1', 'slxml_aircraft2');
```
Understand The Results

The Comparison Tool shows a report that only includes the differences between the two models, not the complete model hierarchies.

Colors indicate if items have been modified (purple items), inserted (blue items), or deleted (yellow items).

Stepping Through Differences

Use the "Next" and "Previous" navigation buttons on the Comparison tab to step through groups of changes in the report. When you do this to select an item:

- The report highlights the corresponding item in the other tree (if it exists)
- Both models display next to the report. The report highlights the corresponding item in the Simulink model, if possible.
You can control highlighting in the highlight toolstrip section.

**Filtering Results**

You can control the type of changes displayed in the comparison report by applying filters. To see the available filters, and whether or not they are applied to the current report, on the Comparison tab, in the Filter section, click the down arrow to expand the filter gallery. Click filter names to toggle whether they are applied. In the Filter section, click Show or Hide to control how the filters are applied.

For example, some information in the Simulink model file is defined as being "nonfunctional". These are changes that are unlikely to change the behavior of the design. Nonfunctional items are hidden by default but you can show them by deselecting the 'Nonfunctional Changes' item in the list of 'Built-In Filters' when the 'Hide' filter toolstrip button is selected.

**Publish Results**

It is possible to create a document containing a record of the changes made to a Simulink model using the "Publish" toolstrip option. Select the 'HTML' publish menu item to create an HTML report. Microsoft Word® reports can also be created by using the 'Word' menu item.

**Merge Changes in Simulink Models**

You can merge the changes between the two Simulink models by clicking the 'Merge Mode' button in the toolstrip. This creates a third file, 'targetFile', which can contain the changes from either the left model (slxml_aircraft1) or right model (slxml_aircraft2). To select which changes you would like in the target file use the radio buttons in the right of the report.

Once you have selected the changes you want in the target model you can save these changes over the right model (slxml_aircraft2) using the 'Save File' toolstrip button.

Note that in order for the tool to automatically connect merged lines, you must merge blocks before lines.

**Further Information**

Compare Simulink Models
Compare and Merge Simulink Models Containing Stateflow

This example shows how to view and merge the changes between two Simulink® models from the comparison report. You can view and merge the Simulink and Stateflow® differences in the resulting report. A Stateflow® license is required to run this example.

This example shows you how to use the comparison report in MATLAB to understand what has changed between two different designs. Some development work has added a new "kickdown" mode to the shift logic. You can use the Simulink Model Comparison to compare the two models and produce a report for exploring and merging the differences.

**Compare The Models**

Setup the example files by running the following command.

```matlab
slxmlcomp.internal.examples.setupStateflowExample();
% Then use visdiff to compare the models.
visdiff('slxml_sfcar_1', 'slxml_sfcar_2');
```
Understand The Results

The Comparison Tool shows a report that only includes the differences between the two models, not the complete model hierarchies.

Colors indicate if items have been modified (purple items), inserted (blue items), or deleted (yellow items).

Stepping Through Differences

Use the "Next" and "Previous" navigation buttons on the Comparison tab to step through groups of changes in the report. When you do this to select an item:

- The report highlights the corresponding item in the other tree (if it exists)
- Both models display next to the report. The report highlights the corresponding item in the Simulink model, if possible.

You can control highlighting in the highlight toolstrip section.
Highlight Differences in the Simulink Models

Click to select the tree node "Out1" in the right hand side tree. Observe the report highlights "Out1" in the report, and the corresponding block in the model "slxml_sfcar_2".

Highlight Differences in a Stateflow Chart

Now browse down the tree to locate the Stateflow charts in the report. Observe the Stateflow chart icon next to the "shift_logic" node. Click "kickdown" in the right tree. The report displays both "shift_logic" Stateflow charts. Observe where the new "kickdown" functionality has been added to one of the charts.

Filtering Results

You can control the type of changes displayed in the comparison report by applying filters. To see the available filters, and whether or not they are applied to the current report, on the Comparison tab, in the Filter section, click the down arrow to expand the filter gallery. Click filter names to toggle whether they are applied. In the Filter section, click Show or Hide to control how the filters are applied.

For example, some information in the Simulink model file is defined as being "nonfunctional". These are changes that are unlikely to change the behavior of the design. Nonfunctional items are hidden by default but you can show them by deselecting the 'Nonfunctional Changes' item in the list of 'Built-In Filters' when the 'Hide' filter toolstrip button is selected.

Merge Changes in Models Containing Stateflow

You can merge the changes between the two Simulink & Stateflow models by clicking the 'Merge Mode' button in the toolstrip. This creates a third file, 'targetFile', which can contain the changes from either the left model or right model. To select which changes you would like in the target file use the radio buttons in the right of the report.

Once you have selected the changes you want in the target model you can save these changes over the right model (slxml_sfcar_2) using the 'Save File' toolstrip button.

Note that in order for the tool to automatically connect merged transitions, you must merge states and junctions before transitions.
Further Information

Compare Simulink Models
Resolve Conflicts with Simulink Three-Way Merge

This example shows you how to use Simulink® Three-Way Merge to resolve conflicts in Simulink models. You can view and merge the Simulink model differences in the resulting report.

Simulink Model Changes in This Example

This example shows you how to use Three-Way Merge to understand and resolve the differences between two conflicting Simulink design changes. The example Simulink project is under source control (Git in this example) and contains changes which conflict with another user's. You will use the Three-Way Model Merge tool to resolve the conflicts. You will examine your local file ("Mine"), the conflicting revision ("Theirs") and the common ancestor of these two files ("Base").

The conflict in this example was caused by the current user trying to merge the changes of another user on the master Git branch (the 'Theirs' changes) into their Task1 branch (containing the "Mine" changes).

Theirs: The pilot model subsystem was updated to use an explicit gain block for the amplitude of the pilot's input commands. The Simulation stop time was also updated.

Mine: The current user changed some configuration settings for the model.

The Merge tool automatically merges nonconflicted differences. Follow the steps below to review the automerge choices, edit if desired, and decide how to resolve any remaining differences. This will then allow you to commit the resolved model to source control.

Set Up Example Simulink Project

Run the following commands to create and open the project.

```
slxmlcomp.internal.examples.setupThreeWayMergeExample();
```

Starting: Simulink

Open Simulink Three-Way Merge

Look for conflicted files in the Simulink Project files view. The slproject_f14 file shows a red warning symbol in the Git column, which indicates a conflict.

1. To see a detailed report of the conflicts, right-click the slproject_f14 file and select 'View Conflicts'.
**View Changes**

The Three-Way Merge tool shows the changes to the two Simulink designs that have caused this file conflict.
• At the top, Theirs, Base and Mine trees show the differences between the conflicting revision, your revision, and the base ancestor of these files.

• Underneath, the Target area shows the file that you will merge changes into. This is a temporary file and will be copied into the Simulink project when you choose to accept the merge result.

2. Examine a difference by clicking a row in one of the trees. The merge tool displays the change for each model in an editor, for example the Simulink Editor or Configuration Parameters dialog box, to the right of the Three-Way Merge window.

3. Choose the models to display with the toolstrip buttons on the Merge tab, in the Highlight section: Top Model or Bottom Model.

**Review Automatic Merges**

The merge tool automatically merges most nonconflicted differences.

4. Examine the first change at the top of the 'Theirs' tree by clicking the row called 'PilotGain'. This node has been merged automatically and you can adjust the automatic choices using the radio buttons in the target pane. If you want, review the other automatic merge choices.

**Resolve Conflicts**

There are two types of difference that require user action.

**Conflict**: The merge tool cannot automatically resolve these differences and you need to choose which design you want in the target file. Look for warnings in the Conflicts column.

**Manual merge**: Some differences must be merged manually in Simulink or ignored. In the Target pane, these items are indicated by an icon with a pencil in the Conflicts column.

5. Select the 'StickCommand_rad' row. This difference requires a manual merge, indicated by a pencil on the icon in the Conflicts column.

Resolve the line change difference in the target file by renaming the line connected to the output of the block named 'Pilot' in the Simulink Editor from 'StickCommand_rad' to 'Pilot Output'.

After resolving the difference save the changes in the Editor, and mark the change resolved using the manual merge control in the Conflicts column.
Note - the merge report does not update to show any changes you make in the Editor.

6. Use the 'Next' toolstrip button to review and resolve the changes until you reach a conflict. This StopTime parameter has been changed by both users and is conflicted. Resolve the conflict using the 'Mine' change. Select the 'Mine' radio button next to StopTime in the Target pane.

**Change Filters**

By default the report hides all nonfunctional changes, such as the repositioning of items.

7. Turn off filters using the Filters toolstrip section to explore all the differences between these designs.

**Accept Changes**

8. After you resolve all filtered and unfiltered changes, click 'Accept & Close'. The merge tool closes the report and the models, accepts the merge result in the target file, and marks the conflict as resolved in the source control tool (Git in this example). Resolved conflicts make it possible to commit changes to source control and continue your work.
Call C Functions Using C Caller Block

This example shows how to use the C Caller block to call your handwritten C functions.

In this example, four custom C functions are defined in my_func.h, and implemented in my_func.c.

The header file and the source file are specified in "Model Configuration Parameters > Simulation Target". Then C functions can be called via C Caller blocks. For example, the block 'add' adds a signal and a constant together, and the block 'timesK' multiplies the signal by a parameter K. C Caller block supports C structure and enumeration types. Use command "Simulink.importExternalCTypes" to import these types into Simulink as matching Simulink.Bus object or Simulink.IntEnumType.
Set up custom C code in model configuration

Add '#include "my_func.h"' to "Configuration Parameters > Simulation Target > Header File" field.

Add 'include' to "Configuration Parameters > Simulation Target > Include directories" field.

Add 'src/my_func.c' to "Configuration Parameters > Simulation Target > Source Files" field.
Bring Custom Image Filter Algorithms as Reusable Blocks in Simulink

This example shows how to bring in C code image filter algorithms using C Caller blocks in a reusable Simulink library.

In this example, three image filter blocks are C Caller blocks in a library model. Their dependent C code is specified in the library model custom code settings.

The image filter C functions are implemented using "Row-major" array layout. Library custom code settings have "Default function array layout" set accordingly to "Row-major".

These library image filter blocks can be reused by adding them to models.
Original Image

Mean Filter

Median Filter

Gaussian Filter
Custom Code and Hand Coded Blocks using the S-function API

This library launches examples of different types of Simulink® S-functions. Simulink S-functions allow you to extend Simulink with new hand coded blocks, interface to custom external code, and create live interfaces to other programs. Models containing S-functions in the following programming languages can be accessed through this demonstration: C, C++, M and Fortran.

Each example contains a model that exercises an S-function block and a link for editing the S-function code. You can also access the main S-function file by using the Edit... button on the S-function block dialog (which is under a mask in some of the models; use the "Look Under Mask" context menu item to access the S-function dialog in this situation).

Additionally, the S-function Builder block is a masked version of the S-function block that includes an easy to use editor that automatically builds an S-function out of a core algorithm that you enter into it along with input, output, and parameter attributes. By default, the S-function Builder also creates a TLC file for use with Simulink® Coder™ and Embedded Coder™.
Inputs Passed by Value or Address to Legacy Functions

This example shows you how to use the Legacy Code Tool to integrate legacy C functions that pass their input arguments by value versus address.

The Legacy Code Tool allows you to:

- Provide the legacy function specification,
- Generate a C-MEX S-function that is used during simulation to call the legacy code, and
- Compile and build the generated S-function for simulation.

Providing the Legacy Function Specification

Functions provided with the Legacy Code Tool take a specific data structure or array of structures as the argument. The data structure is initialized by calling the function `legacy_code()` using 'initialize' as the first input. After initializing the structure, you have to assign its properties to values corresponding to the legacy code being integrated. For detailed help on the properties, call `legacy_code('help')`. The prototypes of the legacy functions being called in this example are:

- `FLT filterV1(const FLT signal, const FLT prevSignal, const FLT gain)`
- `FLT filterV2(const FLT* signal, const FLT prevSignal, const FLT gain)`

where FLT is a typedef to float. The legacy source code is found in the files `your_types.h`, `myfilter.h`, `filterV1.c`, and `filterV2.c`.

Note the difference in the OutputFcnSpec defined in the two structures; the first case specifies that the first input argument is passed by value, while the second case specifies pass by pointer.

defs = [];

% sldemo_sfun_filterV1
def = legacy_code('initialize');
def.SFunctionName = 'sldemo_sfun_filterV1';
def.OutputFcnSpec = 'single y1 = filterV1(single u1, single u2, single p1)';
def.HeaderFiles = {'myfilter.h'};
def.SourceFiles = {'filterV1.c'};
def.IncPaths = {'sldemo_lct_src'};
def.SrcPaths = {'sldemo_lct_src'};
defs = [defs; def];

% sldemo_sfun_filterV2
def = legacy_code('initialize');
def.SFunctionName = 'sldemo_sfun_filterV2';
def.OutputFcnSpec = 'single y1 = filterV2(single u1[1], single u2, single p1)';
def.HeaderFiles = {'myfilter.h'};
def.SourceFiles = {'filterV2.c'};
def.IncPaths = {'sldemo_lct_src'};
def.SrcPaths = {'sldemo_lct_src'};
defs = [defs; def];

### Generating and Compiling an S-Function for Use During Simulation

The function legacy_code() is called again with the first input set to 'generate_for_sim' in order to automatically generate and compile the C-MEX S-function according to the description provided by the input argument 'defs'. This S-function is used to call the legacy functions in simulation. The source code for the S-function is found in the file sldemo_sfun_filterV1.c and sldemo_sfun_filterV2.c.

legacy_code('generate_for_sim', defs);

### Start Compiling sldemo_sfun_filterV1
mex('-IB:\matlab\toolbox\simulink\simdemos\simfeatures\sldemo_lct_src', ...
'-IC:\TEMP\Bdoc19b_1192687_6748\ibF7BE2B\31\tp8ee2474f', ...
'C:\TEMP\Bdoc19b_1192687_6748\ibF7BE2B\31\tp638ce0e2_8a84_4cf7_a2a5_522b52f8da89\filterV1.obj')
Building with 'Microsoft Visual C++ 2017 (C)'.
MEX completed successfully.
mex('sldemo_sfun_filterV1.c', '-IB:\matlab\toolbox\simulink\simdemos\simfeatures\sldemo_lct_src', ...
'-IC:\TEMP\Bdoc19b_1192687_6748\ibF7BE2B\31\tp8ee2474f', ...
'C:\TEMP\Bdoc19b_1192687_6748\ibF7BE2B\31\tp638ce0e2_8a84_4cf7_a2a5_522b52f8da89\filterV1.obj')
Building with 'Microsoft Visual C++ 2017 (C)'.
MEX completed successfully.
### Finish Compiling sldemo_sfun_filterV1
### Exit

### Start Compiling sldemo_sfun_filterV2
mex('-IB:\matlab\toolbox\simulink\simdemos\simfeatures\sldemo_lct_src', ...
'-IC:\TEMP\Bdoc19b_1192687_6748\ibF7BE2B\31\tp8ee2474f', ...
'C:\TEMP\Bdoc19b_1192687_6748\ibF7BE2B\31\tp638ce0e2_8a84_4cf7_a2a5_522b52f8da89\filterV1.obj')
Building with 'Microsoft Visual C++ 2017 (C)'.
MEX completed successfully.
mex('sldemo_sfun_filterV2.c', '-IB:\matlab\toolbox\simulink\simdemos\simfeatures\sldemo_lct_src', ...
'-IC:\TEMP\Bdoc19b_1192687_6748\ibF7BE2B\31\tp8ee2474f', ...
'C:\TEMP\Bdoc19b_1192687_6748\ibF7BE2B\31\tp638ce0e2_8a84_4cf7_a2a5_522b52f8da89\filterV1.obj')
Building with 'Microsoft Visual C++ 2017 (C)'.
MEX completed successfully.
### Finish Compiling sldemo_sfun_filterV2
### Exit
Generating an rtwmakecfg.m File for Code Generation

After the TLC block file is created, the function legacy_code() can be called again with the first input set to 'rtwmakecfg_generate' in order to generate an rtwmakecfg.m file to support code generation through Simulink® Coder™. Generate the rtwmakecfg.m file if the required source and header files for the S-functions are not in the same directory as the S-functions, and you want to add these dependencies in the makefile produced during code generation.

Note: Complete this step only if you are going to simulate the model in accelerated mode.

legacy_code('rtwmakecfg_generate', def);

Generating Masked S-Function Blocks for Calling the Generated S-Functions

After the C-MEX S-function source is compiled, the function legacy_code() can be called again with the first input set to 'slblock_generate' in order to generate masked S-function blocks which are configured to call those S-functions. The blocks are placed in a new model and can be copied to an existing model.

% legacy_code('slblock_generate', defs);

Demoing the Generated Integration with Legacy Code

The model sldemo_lct_filter shows integration with the legacy code. The subsystem TestFilter serves as a harness for the calls to the legacy C functions via the generate S-functions, with unit delays serving to store the previous output values.

open_system('sldemo_lct_filter')
open_system('sldemo_lct_filter/TestFilter')
sim('sldemo_lct_filter');
Inputs Passed by Value or Address to Legacy Functions

First input accessed by value (expression is conserved in the generated code)

First input accessed by address (expression is not conserved in the generated code)
Output Passed by Return Argument from Legacy Functions

This example shows you how to use the Legacy Code Tool to integrate legacy C functions that pass their output as a return argument.

The Legacy Code Tool allows you to:

• Provide the legacy function specification,
• Generate a C-MEX S-function that is used during simulation to call the legacy code, and
• Compile and build the generated S-function for simulation.

Providing the Legacy Function Specification

Functions provided with the Legacy Code Tool take a specific data structure or array of structures as the argument. The data structure is initialized by calling the function legacy_code() using 'initialize' as the first input. After initializing the structure, you have to assign its properties to values corresponding to the legacy code being integrated. For detailed help on the properties, call legacy_code('help'). The prototype of the legacy functions being called in this example is:

FLT gainScalar(const FLT in, const FLT gain)

where FLT is a typedef to float. The legacy source code is found in the files your_types.h, gain.h, and gainScalar.c.

% sldemo_sfun_gain_scalar
def = legacy_code('initialize');
def.SFunctionName = 'sldemo_sfun_gain_scalar';
def.OutputFcnSpec = 'single y1 = gainScalar(single u1, single p1)';
def.HeaderFiles = {'gain.h'};
def.SourceFiles = {'gainScalar.c'};
def.IncPaths = {'sldemo_lct_src'};
def.SrcPaths = {'sldemo_lct_src'};

Generating and Compiling an S-Function for Use During Simulation

The function legacy_code() is called again with the first input set to 'generate_for_sim' in order to automatically generate and compile the C-MEX S-function according to the description provided by the input argument 'def'. This S-function is used to call the legacy
functions in simulation. The source code for the S-function is found in the file sldemo_sfun_gain_scalar.c.

```
legacy_code('generate_for_sim', def);
```

### Start Compiling sldemo_sfun_gain_scalar
```
mex('-IB:\matlab\toolbox\simulink\simdemos\simfeatures\sldemo_lct_src', ...
     '-IC:\TEMP\ibF7BE2B\31\tp7e43a028_ead2_4e4e_9fc6_399479b3b95d\gainScalar.obj')
Building with 'Microsoft Visual C++ 2017 (C)'.
MEX completed successfully.
```

### Finish Compiling sldemo_sfun_gain_scalar
```
### Exit
```

**Generating an rtwmakecfg.m File for Code Generation**

After the TLC block file is created, the function legacy_code() can be called again with the first input set to 'rtwmakecfg_generate' in order to generate an rtwmakecfg.m file to support code generation through Simulink® Coder™. Generate the rtwmakecfg.m file if the required source and header files for the S-functions are not in the same directory as the S-functions, and you want to add these dependencies in the makefile produced during code generation.

Note: Complete this step only if you are going to simulate the model in accelerated mode.

```
legacy_code('rtwmakecfg_generate', def);
```

**Generating a Masked S-Function Block for Calling the Generated S-Function**

After the C-MEX S-function source is compiled, the function legacy_code() can be called again with the first input set to 'slblock_generate' in order to generate a masked S-function block that is configured to call that S-function. The block is placed in a new model and can be copied to an existing model.

```
% legacy_code('slblock_generate', def);
```

**Demoing the Generated Integration with Legacy Code**

The model sldemo_lct_gain shows integration with the legacy code. The subsystem TestGain serves as a harness for the call to the legacy C function via the generate S-function.
if isempty(find_system('SearchDepth',0,'Name','sldemo_lct_gain'))
    open_system('sldemo_lct_gain')
    open_system('sldemo_lct_gain/TestGain')
    sim('sldemo_lct_gain');
end
Fixed Point Signals in Legacy Functions

This example shows you how to use the Legacy Code Tool to integrate legacy C functions that pass their inputs and outputs using signals of fixed point data type.

The Legacy Code Tool allows you to:

• Provide the legacy function specification,
• Generate a C-MEX S-function that is used during simulation to call the legacy code, and
• Compile and build the generated S-function for simulation.

Providing the Legacy Function Specification

Functions provided with the Legacy Code Tool take a specific data structure or array of structures as the argument. The data structure is initialized by calling the function `legacy_code()` using 'initialize' as the first input. After initializing the structure, you have to assign its properties to values corresponding to the legacy code being integrated. For detailed help on the properties, call `legacy_code('help')`. The prototype of the legacy functions being called in this example is:

```c
myFixpt timesS16(const myFixpt in1, const myFixpt in2, const uint8_T fracLength)
```

where `myFixpt` is logically a fixed point data type which is physically a typedef to a 16-bit integer:

```c
myFixpt = Simulink.NumericType;
myFixpt.DataTypeMode = 'Fixed-point: binary point scaling';
myFixpt.Signed = true;
myFixpt.WordLength = 16;
myFixpt.FractionLength = 10;
myFixpt.IsAlias = true;
myFixpt.HeaderFile = 'timesFixpt.h';
```

The legacy source code is found in the files `timesFixpt.h`, and `timesS16.c`.

```matlab
% sldemo_sfun_times_s16
def = legacy_code('initialize');
def.SFunctionName = 'sldemo_sfun_times_s16';
def.OutputFcnSpec = 'myFixpt y1 = timesS16(myFixpt u1, myFixpt u2, uint8 p1)';
def.HeaderFiles = {'timesFixpt.h'};
def.SourceFiles = {'timesS16.c'};
```
def.IncPaths = {'sldemo_lct_src'};
def.SrcPaths = {'sldemo_lct_src'};

Generating and Compiling an S-Function for Use During Simulation

The function legacy_code() is called again with the first input set to 'generate_for_sim' in order to automatically generate and compile the C-MEX S-function according to the description provided by the input argument 'def'. This S-function is used to call the legacy functions in simulation. The source code for the S-function is found in the file sldemo_sfun_times_s16.c.

legacy_code('generate_for_sim', def);

### Start Compiling sldemo_sfun_times_s16
mex('-IB:\matlab\toolbox\simulink\simdemos\simfeatures\sldemo_lct_src', ...
  '-IC:\TEMP\ibF7BE2B\31\tp5a1041aa_0453_456a_a29e_c084606f2b47\timesS16.obj')
Building with 'Microsoft Visual C++ 2017 (C)'.
MEX completed successfully.
mex('sldemo_sfun_times_s16.c', '-IB:\matlab\toolbox\simulink\simdemos\simfeatures\sldemo_lct_src')
Building with 'Microsoft Visual C++ 2017 (C)'.
MEX completed successfully.
### Finish Compiling sldemo_sfun_times_s16
### Exit

Generating an rtwmakecfg.m File for Code Generation

After the TLC block file is created, the function legacy_code() can be called again with the first input set to 'rtwmakecfg_generate' in order to generate an rtwmakecfg.m file to support code generation through Simulink® Coder™. Generate the rtwmakecfg.m file if the required source and header files for the S-functions are not in the same directory as the S-functions, and you want to add these dependencies in the makefile produced during code generation.

Note: Complete this step only if you are going to simulate the model in accelerated mode.

legacy_code('rtwmakecfg_generate', def);

Generating a Masked S-Function Block for Calling the Generated S-Function

After the C-MEX S-function source is compiled, the function legacy_code() can be called again with the first input set to 'slblock_generate' in order to generate a masked S-function block that is configured to call that S-function. The block is placed in a new model and can be copied to an existing model.
Demoing the Generated Integration with Legacy Code

The model sldemo_lct_fixpt_signals shows integration with the legacy code. The subsystem TestFixpt serves as a harness for the call to the legacy C function via the generated S-function, and the scope compares the output of the function with the output of the built-in Simulink® product block; the results are identical.

```matlab
open_system('sldemo_lct_fixpt_signals')
open_system('sldemo_lct_fixpt_signals/TestFixpt')
sim('sldemo_lct_fixpt_signals')

ans =

    Simulink.SimulationOutput:
        yout: [101x2 double]

    SimulationMetadata: [1x1 Simulink.SimulationMetadata]
    ErrorMessage: [0x0 char]
```
Fixed Point Parameters in Legacy Functions

This example shows you how to use the Legacy Code Tool to integrate legacy C functions that pass their inputs and outputs using parameters of fixed point data type.

The Legacy Code Tool allows you to:

- Provide the legacy function specification,
- Generate a C-MEX S-function that is used during simulation to call the legacy code, and
- Compile and build the generated S-function for simulation.

Providing the Legacy Function Specification

Functions provided with the Legacy Code Tool take a specific data structure or array of structures as the argument. The data structure is initialized by calling the function `legacy_code()` using 'initialize' as the first input. After initializing the structure, you have to assign its properties to values corresponding to the legacy code being integrated. For detailed help on the properties, call `legacy_code('help')`. The prototype of the legacy functions being called in this example is:

```c
myFixpt timesS16(const myFixpt in1, const myFixpt in2, const uint8_T fracLength)
```

where `myFixpt` is logically a fixed point data type which is physically a typedef to a 16-bit integer:

```c
def = legacy_code('initialize');
def.SFunctionName = 'myFixpt y1 = timesS16(myFixpt u1, myFixpt p1, uint8 p2)';
def.HeaderFiles   = {'timesFixpt.h'};
def.SourceFiles   = {'timesS16.c'};
```

The legacy source code is found in the files timesFixpt.h, and timesS16.c.
Generating and Compiling an S-Function for Use During Simulation

The function legacy_code() is called again with the first input set to 'generate_for_sim' in order to automatically generate and compile the C-MEX S-function according to the description provided by the input argument 'def'. This S-function is used to call the legacy functions in simulation. The source code for the S-function is found in the file sldemo_sfun_gain_fixpt.c.

```
legacy_code('generate_for_sim', def);
```

### Start Compiling sldemo_sfun_gain_fixpt
```
mex('-IB:\matlab\toolbox\simulink\simdemos\simfeatures\sldemo_lct_src', ... 'B:\matlab\toolbox\simulink\simdemos\simfeatures\sldemo_lct_src	imesS16.c')
Building with 'Microsoft Visual C++ 2017 (C)'.
MEX completed successfully.
mex('sldemo_sfun_gain_fixpt.c', '-IB:\matlab\toolbox\simulink\simdemos\simfeatures\sldemo_lct_src', ... 'C:\TEMP\Bdoc19b_1192687_6748\ibF7BE2B\31	pe6abc9c6_fcb1_49fe_9364_3c223ca32404	imesS16.obj')
Building with 'Microsoft Visual C++ 2017 (C)'.
MEX completed successfully.
### Finish Compiling sldemo_sfun_gain_fixpt
### Exit
```

Generating an rtwmakecfg.m File for Code Generation

After the TLC block file is created, the function legacy_code() can be called again with the first input set to 'rtwmakecfg_generate' in order to generate an rtwmakecfg.m file to support code generation through Simulink® Coder™. Generate the rtwmakecfg.m file if the required source and header files for the S-functions are not in the same directory as the S-functions, and you want to add these dependencies in the makefile produced during code generation.

Note: Complete this step only if you are going to simulate the model in accelerated mode.

```
legacy_code('rtwmakecfg_generate', def);
```

Generating a Masked S-Function Block for Calling the Generated S-Function

After the C-MEX S-function source is compiled, the function legacy_code() can be called again with the first input set to 'slblock_generate' in order to generate a masked S-function block that is configured to call that S-function. The block is placed in a new model and can be copied to an existing model.
Demoing the Generated Integration with Legacy Code

The model sldemo_lct_fixpt_params shows integration with the legacy code. The subsystem TestFixpt serves as a harness for the call to the legacy C function via the generated S-function.

```matlab
open_system('sldemo_lct_fixpt_params')
open_system('sldemo_lct_fixpt_params/TestFixpt')
sim('sldemo_lct_fixpt_params')
```

ans =

```
Simulink.SimulationOutput:
    yout: [101x4 double]

    SimulationMetadata: [1x1 Simulink.SimulationMetadata]
    ErrorMessage: [0x0 char]
```
Inlined parameters (InlineParameters ON + Auto storage class) => numeric value inlined

```
myFixpt y1 = timesS15(myFixpt u1, myFixpt p1, uint8 p2)
```

```
sldeono_sfun_gain_fixpt
```

Double-precision (context-sensitive) parameters => tunable parameter inherits data type from run-time parameter

```
myFixpt y1 = timesS15(myFixpt u1, myFixpt p1, uint8 p2)
```

```
sldeono_sfun_gain_fixpt1
```

Tunable parameter with myFixpt data type specification => parameter is not cast

```
myFixpt y1 = timesS15(myFixpt u1, myFixpt p1, uint8 p2)
```

```
sldeono_sfun_gain_fixpt2
```

Tunable parameter with explicit data type specification (uint8) => parameter is cast to run-time parameter data type

```
myFixpt y1 = timesS15(myFixpt u1, myFixpt p1, uint8 p2)
```

```
sldeono_sfun_gain_fixpt3
```
Lookup Tables Implemented in Legacy Functions

This example shows you how to use the Legacy Code Tool to integrate legacy C functions that implement N-dimensional table lookups.

The Legacy Code Tool allows you to:

- Provide the legacy function specification,
- Generate a C-MEX S-function that is used during simulation to call the legacy code, and
- Compile and build the generated S-function for simulation.

Providing the Legacy Function Specification

Functions provided with the Legacy Code Tool take a specific data structure or array of structures as the argument. The data structure is initialized by calling the function legacy_code() using 'initialize' as the first input. After initializing the structure, you have to assign its properties to values corresponding to the legacy code being integrated. For detailed help on the properties, call legacy_code('help'). The prototype of the legacy function being called in this example is:

```
FLT directLookupTableND(const FLT *tableND, const UINT32 nbDims, const UINT32 *tableDims, const UINT32 *tableIdx)
```

where FLT is a typedef to float, and UINT32 is a typedef to unsigned int32. The legacy source code is found in the files your_types.h, lookupTable.h, and directLookupTableND.c.

```matlab
defs = [];
evalin('base','load sldemo_lct_data.mat')

% sldemo_sfun_dlut3D
def = legacy_code('initialize');
def.SFunctionName = 'sldemo_sfun_dlut3D';
def.OutputFcnSpec = 'single y1 = DirectLookupTable3D(single p1[[[]]], uint32 p2[3], uint32 u1[3])';
def.HeaderFiles = {'lookupTable.h'};
def.SourceFiles = {'directLookupTableND.c'};
def.IncPaths = {'sldemo_lct_src'};
def.SrcPaths = {'sldemo_lct_src'};
defs = [defs; def];

% sldemo_sfun_dlut4D
def = legacy_code('initialize');
```
def.SFunctionName = 'sldemo_sfun_dlut4D';
def.OutputFcnSpec = 'single y1 = DirectLookupTable4D(single p1[][],[], uint32 p2[4], uint32 u1[4]);';
def.HeaderFiles   = {'lookupTable.h'};
def.SourceFiles   = {'directLookupTableND.c'};
def.IncPaths      = {'sldemo_lct_src'};
def.SrcPaths      = {'sldemo_lct_src'};
defs = [defs; def];

Generating and Compiling an S-Function for Use During Simulation

The function legacy_code() is called again with the first input set to 'generate_for_sim' in order to automatically generate and compile the C-MEX S-function according to the description provided by the input argument 'defs'. This S-function is used to call the legacy functions in simulation. The source code for the S-function is found in the file sldemo_sfun_dlut3D.c and sldemo_sfun_dlut4D.c.

legacy_code('generate_for_sim', defs);

### Start Compiling sldemo_sfun_dlut3D
mex('-IB:\matlab\toolbox\simulink\simdemos\simfeatures\sldemo_lct_src', '-IC:\TEMP\Bdoc19b_1192687_6748\ibF7BE2B\31\tp257844ff_512f_4f51_a47c_b62f991229f9\directLookupTableND.obj')
Building with 'Microsoft Visual C++ 2017 (C)'.
MEX completed successfully.
mex('sldemo_sfun_dlut3D.c', '-IB:\matlab\toolbox\simulink\simdemos\simfeatures\sldemo_lct_src', '-IC:\TEMP\Bdoc19b_1192687_6748\ibF7BE2B\31\tp257844ff_512f_4f51_a47c_b62f991229f9\directLookupTableND.obj')
Building with 'Microsoft Visual C++ 2017 (C)'.
MEX completed successfully.
### Finish Compiling sldemo_sfun_dlut3D
### Exit

### Start Compiling sldemo_sfun_dlut4D
mex('-IB:\matlab\toolbox\simulink\simdemos\simfeatures\sldemo_lct_src', '-IC:\TEMP\Bdoc19b_1192687_6748\ibF7BE2B\31\tp257844ff_512f_4f51_a47c_b62f991229f9\directLookupTableND.obj')
Building with 'Microsoft Visual C++ 2017 (C)'.
MEX completed successfully.
mex('sldemo_sfun_dlut4D.c', '-IB:\matlab\toolbox\simulink\simdemos\simfeatures\sldemo_lct_src', '-IC:\TEMP\Bdoc19b_1192687_6748\ibF7BE2B\31\tp257844ff_512f_4f51_a47c_b62f991229f9\directLookupTableND.obj')
Building with 'Microsoft Visual C++ 2017 (C)'.
MEX completed successfully.
### Finish Compiling sldemo_sfun_dlut4D
### Exit

Generating an rtwmakecfg.m File for Code Generation

After the TLC block file is created, the function legacy_code() can be called again with the first input set to 'rtwmakecfg_generate' in order to generate an rtwmakecfg.m file to support code generation through Simulink® Coder™. Generate the rtwmakecfg.m file if
the required source and header files for the S-functions are not in the same directory as the S-functions, and you want to add these dependencies in the makefile produced during code generation.

Note: Complete this step only if you are going to simulate the model in accelerated mode.

legacy_code('rtwmakecfg_generate', def);

**Generating a Masked S-Function Block for Calling the Generated S-Function**

After the C-MEX S-function source is compiled, the function legacy_code() can be called again with the first input set to 'slblock_generate' in order to generate a masked S-function block that is configured to call that S-function. The block is placed in a new model and can be copied to an existing model.

% legacy_code('slblock_generate', defs);

**Demoing the Generated Integration with Legacy Code**

The model sldemo_lct_lut shows integration with the legacy code. The subsystem TestFixpt serves as a harness for the call to the legacy C function, and the Display blocks compare the output of the function with the output of the built-in Simulink® lookup blocks; the results are identical.

open_system('sldemo_lct_lut')
opensystem('sldemo_lct_lut/TestLut1')
sim('sldemo_lct_lut');
LookUp Table are defined as:

```plaintext
>> LUT3D = single(reshape([1:4*5*6], [4 5 6]));
>> LUT4D = single(reshape([1:4*5*6*2], [4 5 6 2]))
```

**3D table**

```
uint32 (3)
```

```
single y1 = DirectLookupTable3D(single p1[3], uint32 p2[3], uint32 u1[3])
```

```
sldemo_sfunدلut3D
```

**4D table**

```
uint32 (4)
```

```
single y1 = DirectLookupTable4D(single p1[4], uint32 p2[4], uint32 u1[4])
```

```
sldemo_sfunدلut4D
```
Start and Terminate Actions Within Legacy Functions

This example shows you how to use the Legacy Code Tool to integrate legacy C functions that have start and terminate actions.

The Legacy Code Tool allows you to:

• Provide the legacy function specification,
• Generate a C-MEX S-function that is used during simulation to call the legacy code, and
• Compile and build the generated S-function for simulation.

Providing the Legacy Function Specification

Functions provided with the Legacy Code Tool take a specific data structure or array of structures as the argument. The data structure is initialized by calling the function `legacy_code()` using the 'initialize' first input. After initializing the structure, you have to assign its properties to values corresponding to the legacy code being integrated. For detailed help on the properties, call `legacy_code('help')`. The prototypes of the legacy functions being called in this example are:

• `void initFaultCounter(unsigned int *counter)`
• `void openLogFile(void **fid)`
• `void incAndLogFaultCounter(void *fid, unsigned int *counter, double time)`
• `void closeLogFile(void **fid)`

The legacy source code is found in the files `your_types.h`, `fault.h`, and `fault.c`.

```matlab
% sldemo_sfun_fault
def = legacy_code('initialize');
def.SFunctionName                = 'sldemo_sfun_fault';
def.InitializeConditionsFcnSpec  = 'initFaultCounter(uint32 work2[1])';
def.StartFcnSpec                 = 'openLogFile(void **work1)';
def.OutputFcnSpec                = 'incAndLogFaultCounter(void *work1, uint32 work2[1], double u1)';
def.TerminateFcnSpec             = 'closeLogFile(void **work1)';
def.HeaderFiles                  = {'fault.h'};
def.SourceFiles                  = {'fault.c'};
def.IncPaths                     = {'sldemo_lct_src'};
def.SrcPaths                     = {'sldemo_lct_src'};
def.Options.useTlcWithAccel      = false;
```
Generating and Compiling an S-Function for Use During Simulation

The function legacy_code() is called again with the first input set to 'generate_for_sim' in order to automatically generate and compile the C-MEX S-function according to the description provided by the input argument 'def'. This S-function is used to call the legacy functions in simulation. The source code for the S-function is found in the file sldemo_sfun_fault.c.

legacy_code('generate_for_sim', def);

### Start Compiling sldemo_sfun_fault
mex('-IB:\matlab\toolbox\simulink\simdemos\simfeatures\sldemo_lct_src', ...
     '-IC:\TEMP\Bdoc19b_1192687_6748\ibF7BE2B\31\tp8ee2474f', ...
     'C:\TEMP\Bdoc19b_1192687_6748\ibF7BE2B\31\tp3a5412e2_3d0c_4c3e_9443_291726371bef\fault.obj')
Building with 'Microsoft Visual C++ 2017 (C)'.
MEX completed successfully.

### Finish Compiling sldemo_sfun_fault
### Exit

Generating an rtwmakecfg.m File for Code Generation

After the TLC block file is created, the function legacy_code() can be called again with the first input set to 'rtwmakecfg_generate' in order to generate an rtwmakecfg.m file to support code generation through Simulink® Coder™. Generate the rtwmakecfg.m file if the required source and header files for the S-functions are not in the same directory as the S-functions, and you want to add these dependencies in the makefile produced during code generation.

Note: Complete this step only if you are going to simulate the model in accelerated mode.

legacy_code('rtwmakecfg_generate', def);

Generating a Masked S-Function Block for Calling the Generated S-Function

After the C-MEX S-function source is compiled, the function legacy_code() can be called again with the first input set to 'slblock_generate' in order to generate a masked S-function block that is configured to call that S-function. The block is placed in a new model and can be copied to an existing model.

% legacy_code('slblock_generate', def);
Demoing the Generated Integration with Legacy Code

The model sldemo_lct_start_term shows integration with the legacy code. The subsystem TestFixpt serves as a harness for the call to the legacy C function, and the scope compares the output of the function with the output of the built-in Simulink® product block; the results should be identical.

```matlab
open_system('sldemo_lct_start_term')
only_system('sldemo_lct_start_term/TestFault')
sim('sldemo_lct_start_term');
```
This demonstration illustrates how the Legacy Code Tool is used to call legacy functions that have start and terminate actions.

- run the model and edit the generated logging file 'sldemo_lct_fault.log'

To view the sources for this demonstration, click the links below.

- open legacy files fault.h and fault.c
- open registration script sldemo_lct_start_term_script.m
- open generated S-Function sldemo_sfun_fault.c

Copyright 1990-2012 The MathWorks, Inc.
Using Buses with Legacy Functions Having Structure Arguments

This example shows you how to use the Legacy Code Tool to integrate legacy C functions with structure arguments using Simulink® buses.

The Legacy Code Tool allows you to:

- Provide the legacy function specification,
- Generate a C-MEX S-function that is used during simulation to call the legacy code, and
- Compile and build the generated S-function for simulation.

Providing the Legacy Function Specification

Functions provided with the Legacy Code Tool take a specific data structure or array of structures as the argument. The data structure is initialized by calling the function `legacy_code()` using 'initialize' as the first input. After initializing the structure, you have to assign its properties to values corresponding to the legacy code being integrated. For detailed help on the properties, call `legacy_code('help')`. The prototype of the legacy functions being called in this example is:

```
counterbusFcn(COUNTERBUS *u1, int32_T u2, COUNTERBUS *y1, int32_T *y2)
```

where COUNTERBUS is a struct typedef defined in counterbus.h and implemented with a Simulink.Bus object in the base workspace. The legacy source code is found in the files counterbus.h, and counterbus.c.

```
evalin('base','load sldemo_lct_data.mat')
```

```
% sldemo_sfun_counterbus
def = legacy_code('initialize');
def.SFunctionName = 'sldemo_sfun_counterbus';
def.OutputFcnSpec = 'void counterbusFcn(COUNTERBUS u1[1], int32 u2, COUNTERBUS y1[1], int32 y2[1])';
def/HeaderFiles = {'counterbus.h'};
def/SourceFiles = {'counterbus.c'};
def/IncPaths = {'sldemo_lct_src'};
def/SrcPaths = {'sldemo_lct_src'};
```
Generating and Compiling an S-Function for Use During Simulation

The function legacy_code() is called again with the first input set to 'generate_for_sim' in order to automatically generate and compile the C-MEX S-function according to the description provided by the input argument 'def'. This S-function is used to call the legacy functions in simulation. The source code for the S-function is found in the file sldemo_sfund_counterbus.c.

```plaintext
legacy_code('generate_for_sim', def);
```

### Start Compiling sldemo_sfund_counterbus
```
mex('-IB:\matlab\toolbox\simulink\simdemos\simfeatures\sldemo_lct_src', '-IC:\\TEMP\\counterbus.obj')
Building with 'Microsoft Visual C++ 2017 (C)'.
MEX completed successfully.
mex('sldemo_sfund_counterbus.c', '-IB:\matlab\toolbox\simulink\simdemos\simfeatures\counterbus.obj')
Building with 'Microsoft Visual C++ 2017 (C)'.
MEX completed successfully.
### Finish Compiling sldemo_sfund_counterbus
### Exit

Generating an rtwmakecfg.m File for Code Generation

After the TLC block file is created, the function legacy_code() can be called again with the first input set to 'rtwmakecfg_generate' in order to generate an rtwmakecfg.m file to support code generation through Simulink® Coder™. Generate the rtwmakecfg.m file if the required source and header files for the S-functions are not in the same directory as the S-functions, and you want to add these dependencies in the makefile produced during code generation.

Note: Complete this step only if you are going to simulate the model in accelerated mode.

```plaintext
legacy_code('rtwmakecfg_generate', def);
```

Generating a Masked S-Function Block for Calling the Generated S-Function

After the C-MEX S-function source is compiled, the function legacy_code() can be called again with the first input set to 'slblock_generate' in order to generate a masked S-function block that is configured to call that S-function. The block is placed in a new model and can be copied to an existing model.

```plaintext
% legacy_code('slblock_generate', def);
```
**Demoing the Generated Integration with Legacy Code**

The model sldemo_lct_bus shows integration with the legacy code. The subsystem TestCounter serves as a harness for the call to the legacy C function.

```matlab
open_system('sldemo_lct_bus')
open_system('sldemo_lct_bus/TestCounter')
sim('sldemo_lct_bus')
```

```matlab
ans =

Simulink.SimulationOutput:
  ScopeDataA: [301x4 double]
  ScopeDataA1: [301x4 double]
  tout: [301x1 double]

  SimulationMetadata: [1x1 Simulink.SimulationMetadata]
  ErrorMessage: [0x0 char]
```
Using Buses with Legacy Functions Having Structure Arguments
The Legacy Code Tool allows you to:

- Provide the legacy function specification,
- Generate a C-MEX S-function that is used during simulation to call the legacy code, and
- Compile and build the generated S-function for simulation.

Providing the Legacy Function Specification

Functions provided with the Legacy Code Tool take a specific data structure or array of structures as the argument. The data structure is initialized by calling the function `legacy_code()` using 'initialize' as the first input. After initializing the structure, you have to assign its properties to values corresponding to the legacy code being integrated. For detailed help on the properties, call `legacy_code('help'). The prototypes of the legacy functions being called in this example are:

- `void mat_add(real_T *u1, real_T *u2, int32_T nbRows, int32_T nbCols, real_T *y1)`
- `void mat_mult(real_T *u1, real_T *u2, int32_T nbRows1, int32_T nbCols1, int32_T nbCols2, real_T *y1)`

where `real_T` is a typedef to double, and `int32_T` is a typedef to a 32-bit integer. The legacy source code is found in the files `mat_ops.h`, and `mat_ops.c`.

```matlab
defs = [];

% sldemo_sfun_mat_add
def = legacy_code('initialize');
def.SFunctionName = 'sldemo_sfun_mat_add';
def.OutputFcnSpec = ['void mat_add(double u1[][], double u2[][],', ' ...
                     'int32 u3, int32 u4, double y1[size(u1,1)][size(u1,2)])'];
def.HeaderFiles = {'mat_ops.h'};
def.SourceFiles = {'mat_ops.c'};
def.IncPaths = {'sldemo_lct_src'};
def.SrcPaths = {'sldemo_lct_src'};
defs = [defs; def];```
% sldemo_sfun_mat_mult
def = legacy_code('initialize');
def.SFunctionName = 'sldemo_sfun_mat_mult';
def.OutputFcnSpec = 
    ['void mat_mult(double u1[p1][p2], double u2[p2][p3], ...
     int32 p1, int32 p2, int32 p3, double y1[p1][p3])'];
def.HeaderFiles   = {'mat_ops.h'};
def.SourceFiles   = {'mat_ops.c'};
def.IncPaths      = {'sldemo_lct_src'};
def.SrcPaths      = {'sldemo_lct_src'};
defs = [defs; def];

Generating and Compiling an S-Function for Use During Simulation

The function legacy_code() is called again with the first input set to 'generate_for_sim' in order to automatically generate and compile the C-MEX S-function according to the description provided by the input argument 'defs'. This S-function is used to call the legacy functions in simulation. The source code for the S-function is found in the file sldemo_sfun_mat_add.c and sldemo_sfun_mat_mult.c.

legacy_code('generate_for_sim', defs);

### Start Compiling sldemo_sfun_mat_add
mex('-IB:\matlab\toolbox\simulink\simdemos\simfeatures\sldemo_lct_src', ...
'B:\matlab\toolbox\simulink\ simdemos\simfeatures\sldemo_lct_src\mat_ops.c')
Building with 'Microsoft Visual C++ 2017 (C)'.
MEX completed successfully.
mex('sldemo_sfun_mat_add.c', '-IB:\matlab\toolbox\simulink\simdemos\simfeatures\sldemo_lct_src', ...
'C:\TEMP\Bdoc19b_1192687_6748\ibF7BE2B\31\tp8ee2474f', 'C:\TEMP\Bdoc19b_1192687_6748\ibF7BE2B\31\tpbfd387a2_31b7_4f6c_b8ff_a1ba6c7557d1\mat_ops.obj')
Building with 'Microsoft Visual C++ 2017 (C)'.
MEX completed successfully.
### Finish Compiling sldemo_sfun_mat_add
### Exit

### Start Compiling sldemo_sfun_mat_mult
mex('-IB:\matlab\toolbox\simulink\simdemos\simfeatures\sldemo_lct_src', ...
'B:\matlab\toolbox\simulink\simdemos\simfeatures\sldemo_lct_src\mat_ops.c')
Building with 'Microsoft Visual C++ 2017 (C)'.
MEX completed successfully.
mex('sldemo_sfun_mat_mult.c', '-IB:\matlab\toolbox\simulink\simdemos\simfeatures\sldemo_lct_src', ...
'C:\TEMP\Bdoc19b_1192687_6748\ibF7BE2B\31\tp8ee2474f', 'C:\TEMP\Bdoc19b_1192687_6748\ibF7BE2B\31\tpbfd387a2_31b7_4f6c_b8ff_a1ba6c7557d1\mat_ops.obj')
Building with 'Microsoft Visual C++ 2017 (C)'.
MEX completed successfully.
### Finish Compiling sldemo_sfun_mat_mult
### Exit
Generating an `rtwmakecfg.m` File for Code Generation

After the TLC block file is created, the function `legacy_code()` can be called again with the first input set to 'rtwmakecfg_generate' in order to generate an `rtwmakecfg.m` file to support code generation through Simulink® Coder™. Generate the `rtwmakecfg.m` file if the required source and header files for the S-functions are not in the same directory as the S-functions, and you want to add these dependencies in the makefile produced during code generation.

Note: Complete this step only if you need to simulate the model in accelerated mode.

```matlab
legacy_code('rtwmakecfg_generate', def);
```

Generating Masked S-Function Blocks for Calling the Generated S-Functions

After the C-MEX S-function source is compiled, the function `legacy_code()` can be called again with the first input set to 'slblock_generate' in order to generate masked S-function blocks which are configured to call those S-functions. The blocks are placed in a new model and can be copied to an existing model.

```matlab
% legacy_code('slblock_generate', defs);
```

Demoing the Generated Integration with Legacy Code

The model `sldemo_lct_inherit_dims` shows integration with the legacy code. The subsystem `TestMatOps` serves as a harness for the calls to the legacy C functions, with unit delays serving to store the previous output values.

```matlab
open_system('sldemo_lct_inherit_dims')
open_system('sldemo_lct_inherit_dims/TestMatOps')
sim('sldemo_lct_inherit_dims');
```
Input/Output matrix dimensions are related to the parameter’s values

Input matrices are dynamically sized
the Output matrix dimensions are related to the Input matrices dimensions
C++ Object Methods as Legacy Functions

This example shows you how to use the Legacy Code Tool to integrate legacy C++ object methods.

The Legacy Code Tool allows you to:

• Provide the legacy function specification,
• generate a C++ MEX S-function that is used during simulation to call the legacy code, and
• Compile and build the generated S-function for simulation.

Providing the Legacy Function Specification

Functions provided with the Legacy Code Tool take a specific data structure or array of structures as the argument. The data structure is initialized by calling the function `legacy_code()` using 'initialize' as the first input. After initializing the structure, you have to assign its properties to values corresponding to the legacy code being integrated. For detailed help on the properties, call `legacy_code('help')`. The definition of the legacy C++ class being used in this example is:

```cpp
class adder {
    private:
        int int_state;
    public:
        adder();
        int add_one(int increment);
        int get_val();
};
```

The legacy source code is found in the files `adder_cpp.h` and `adder_cpp.cpp`.

```matlab
% sldemo_sfun_adder_cpp
def = legacy_code('initialize');
def.SFunctionName = 'sldemo_sfun_adder_cpp';
def.StartFcnSpec = 'createAdder()';
def.OutputFcnSpec = 'int32 y1 = adderOutput(int32 u1)';
def.TerminateFcnSpec = 'deleteAdder()';
def.HeaderFiles   = {'adder_cpp.h'};
def.SourceFiles   = {'adder_cpp.cpp'};
def.IncPaths      = {'sldemo_lct_src'};
def.SrcPaths      = {'sldemo_lct_src'};
```
def.Options.language = 'C++';
def.Options.useTlcWithAccel = false;

**Generating and Compiling an S-Function for Use During Simulation**

The function legacy_code() is called again with the first input set to 'generate_for_sim' in order to automatically generate and compile the C-MEX S-function according to the description provided by the input argument 'def'. This S-function is used to call the legacy functions in simulation. The source code for the S-function is found in the file sldemo_sfun_adder_cpp.cpp.

legacy_code('generate_for_sim', def);

```matlab
### Start Compiling sldemo_sfun_adder_cpp
mex('-IB:\matlab\toolbox\simulink\simdemos\simfeatures\sldemo_lct_src', '-IC:\TEMP\Bdoc19b_1192687_6748\ibF7BE2B\31\tp6cee69e1_5be6_4733_b44f_2e85cefa9cd4\adder_cpp.obj')
Building with 'Microsoft Visual C++ 2017'.
MEX completed successfully.
mex('sldemo_sfun_adder_cpp.cpp', '-IB:\matlab\toolbox\simulink\simdemos\simfeatures\sldemo_lct_src', '-IC:\TEMP\Bdoc19b_1192687_6748\ibF7BE2B\31\tp6cee69e1_5be6_4733_b44f_2e85cefa9cd4\adder_cpp.obj')
Building with 'Microsoft Visual C++ 2017'.
MEX completed successfully.
### Finish Compiling sldemo_sfun_adder_cpp
### Exit
```

**Generating an rtwmakecfg.m File for Code Generation**

After the TLC block file is created, the function legacy_code() can be called again with the first input set to 'rtwmakecfg_generate' in order to generate an rtwmakecfg.m file to support code generation through Simulink® Coder™. Generate the rtwmakecfg.m file if the required source and header files for the S-functions are not in the same directory as the S-functions, and you want to add these dependencies in the makefile produced during code generation.

Note: Complete this step only if you are going to simulate the model in accelerated mode.

legacy_code('rtwmakecfg_generate', def);

**Generating a Masked S-Function Block for Calling the Generated S-Function**

After the C-MEX S-function source is compiled, the function legacy_code() can be called again with the first input set to 'slblock_generate' in order to generate a masked S-function block that is configured to call that S-function. The block is placed in a new model and can be copied to an existing model.
Demoing the Generated Integration with Legacy Code

The model sldemo_lct_cpp shows integration with the legacy code.

open_system('sldemo_lct_cpp')
sim('sldemo_lct_cpp');
Simulink Featured Examples
Persistent Memory Within Legacy Functions

This example shows you how to use the Legacy Code Tool to integrate legacy C functions with instance specific persistent memory.

The Legacy Code Tool allows you to:

- Provide the legacy function specification,
- Generate a C-MEX S-function that is used during simulation to call the legacy code, and
- Compile and build the generated S-function for simulation.

Providing the Legacy Function Specification

Functions provided with the Legacy Code Tool take a specific data structure or array of structures as the argument. The data structure is initialized by calling the function legacy_code() using 'initialize' as the first input. After initializing the structure, you have to assign its properties to values corresponding to the legacy code being integrated. For detailed help on the properties, call legacy_code('help'). The prototype of the legacy functions being called in this example are:

```c
void memory_bus_init(COUNTERBUS *mem, int32_T upper_sat, int32_T lower_sat);

void memory_bus_step(COUNTERBUS *input, COUNTERBUS *mem, COUNTERBUS *output);
```

where mem is an instance specific persistent memory for applying a one integration step delay. COUNTERBUS is a struct typedef defined in counterbus.h and implemented with a Simulink.Bus object in the base workspace. The legacy source code is found in the files memory_bus.h, and memory_bus.c.

```matlab
evalin('base','load sldemo_lct_data.mat')
% sldemo_sfun_work
def = legacy_code('initialize');
def.SFunctionName = 'sldemo_sfun_work';
def.InitializeConditionsFcnSpec = 'void memory_bus_init(COUNTERBUS work1[1], int32 p1, int32 p2)';
def.OutputFcnSpec = 'void memory_bus_step(COUNTERBUS u1[1], COUNTERBUS work1[1], COUNTERBUS y1[1])';
def.HeaderFiles   = {'memory_bus.h'};
def.SourceFiles   = {'memory_bus.c'};
def.IncPaths      = {'sldemo_lct_src'};
def.SrcPaths      = {'sldemo_lct_src'};
```
Generating and Compiling an S-Function for Use During Simulation

The function legacy_code() is called again with the first input set to 'generate_for_sim' in order to automatically generate and compile the C-MEX S-function according to the description provided by the input argument 'def'. This S-function is used to call the legacy functions in simulation. The source code for the S-function is found in the file sldemo_sfunk_work.c.

legacy_code('generate_for_sim', def);

### Start Compiling sldemo_sfunk_work
mex('-IB:\matlab\toolbox\simulink\simdemos\simfeatures\sldemo_lct_src', '-IC:\TEMP\Bdoc19b_1192687_6748\ibF7BE2B\31\tp8ee2474f', 'C:\TEMP\Bdoc19b_1192687_6748\ibF7BE2B\31\tp239e44e9_5c6c_4c62_a905_2288849eac91\memory_bus.obj')
Building with 'Microsoft Visual C++ 2017 (C)'.
MEX completed successfully.

### Finish Compiling sldemo_sfunk_work
### Exit

Generating an rtwmakecfg.m File for Code Generation

After the TLC block file is created, the function legacy_code() can be called again with the first input set to 'rtwmakecfg_generate' in order to generate an rtwmakecfg.m file to support code generation through Simulink® Coder™. Generate the rtwmakecfg.m file if the required source and header files for the S-functions are not in the same directory as the S-functions, and you want to add these dependencies in the makefile produced during code generation.

Note: Complete this step only if you are going to simulate the model in accelerated mode.

legacy_code('rtwmakecfg_generate', def);

Generating a Masked S-Function Block for Calling the Generated S-Function

After the C-MEX S-function source is compiled, the function legacy_code() can be called again with the first input set to 'slblock_generate' in order to generate a masked S-function block that is configured to call that S-function. The block is placed in a new model and can be copied to an existing model.

% legacy_code('slblock_generate', def);
Demoing the Generated Integration with Legacy Code

The model sldemo_lct_work shows integration with the legacy code. The subsystem memory_bus serves as a harness for the call to the legacy C function.

```matlab
open_system('sldemo_lct_work')
open_system('sldemo_lct_work/memory_bus')
sim('sldemo_lct_work')
```

ans =

```
Simulink.SimulationOutput:
    ScopeDataA: [11x3 double]
        tout: [11x1 double]

    SimulationMetadata: [1x1 Simulink.SimulationMetadata]
        ErrorMessage: [0x0 char]
```

This legacy function apply a one integration step delay. The output is the previous input value. The parameters P1 and P2 set the initial values of the sub structure fields "upper_saturation_limit" and "lower_saturation_limit"
Multi-Dimensional Signals in Legacy Functions

This example shows you how to use the Legacy Code Tool to integrate legacy C functions with Multi-Dimensional Signals.

The Legacy Code Tool allows you to:

- Provide the legacy function specification,
- Generate a C-MEX S-function that is used during simulation to call the legacy code, and
- Compile and build the generated S-function for simulation.

Providing the Legacy Function Specification

Functions provided with the Legacy Code Tool take a specific data structure or array of structures as the argument. The data structure is initialized by calling the function legacy_code() using 'initialize' as the first input. After initializing the structure, you have to assign its properties to values corresponding to the legacy code being integrated. For detailed help on the properties, call legacy_code('help'). The prototype of the legacy functions being called in this example are:

```c
void array3d_add(real_T *y1, real_T *u1, real_T *u2, int32_T nbRows, int32_T nbCols, int32_T nbPages);
```

where real_T is a typedef to double, and int32_T is a typedef to a 32-bit integer. The legacy source code is found in the files ndarray_ops.h, and ndarray_ops.c.

```matlab
% sldemo_sfun_ndarray_add
def = legacy_code('initialize');
def.SFunctionName = 'sldemo_sfun_ndarray_add';
def.OutputFcnSpec = ['void array3d_add(double y1[size(u1,1)][size(u1,2)][size(u1,3)], ',
    'double u1[][][], double u2[][][], ', ...
    'int32 size(u1,1), int32 size(u1,2), int32 size(u1,3))'];
def.HeaderFiles = {'ndarray_ops.h'};
def.SourceFiles = {'ndarray_ops.c'};
def.IncPaths = {'sldemo_lct_src'};
def.SrcPaths = {'sldemo_lct_src'};
```

where y1 is a 3-D output signal of same dimensions as the 3-D input signal u1. Note that the last 3 arguments passed to the legacy function correspond to the number of element in each dimension of the 3-D input signal u1.
Generating and Compiling an S-Function for Use During Simulation

The function legacy_code() is called again with the first input set to 'generate_for_sim' in order to automatically generate and compile the C-MEX S-function according to the description provided by the input argument 'def'. This S-function is used to call the legacy functions in simulation. The source code for the S-function is found in the file sldemo_sfun_ndarray_add.c.

legacy_code('generate_for_sim', def);

### Start Compiling sldemo_sfun_ndarray_add
mex('-IB:\matlab\toolbox\simulink\simdemos\simfeatures\sldemo_lct_src', '-IC:\TEMP\', ... 'B:\matlab\toolbox\simulink\simdemos\simfeatures\sldemo_lct_src\ndarray_ops.c')
Building with 'Microsoft Visual C++ 2017 (C)'.
MEX completed successfully.
  mex('sldemo_sfun_ndarray_add.c', '-IB:\matlab\toolbox\simulink\simdemos\simfeatures\sldemo_lct_src\ndarray_ops.obj', '-IC:\TEMP\', ... 'B:\matlab\toolbox\simulink\simdemos\simfeatures\sldemo_lct_src\ndarray_ops.obj')
Building with 'Microsoft Visual C++ 2017 (C)'.
MEX completed successfully.
### Finish Compiling sldemo_sfun_ndarray_add
### Exit

Generating an rtwmakecfg.m File for Code Generation

After the TLC block file is created, the function legacy_code() can be called again with the first input set to 'rtwmakecfg_generate' in order to generate an rtwmakecfg.m file to support code generation through Simulink® Coder™. Generate the rtwmakecfg.m file if the required source and header files for the S-functions are not in the same directory as the S-functions, and you want to add these dependencies in the makefile produced during code generation.

Note: Complete this step only if you are going to simulate the model in accelerated mode.

legacy_code('rtwmakecfg_generate', def);

Generating a Masked S-Function Block for Calling the Generated S-Function

After the C-MEX S-function source is compiled, the function legacy_code() can be called again with the first input set to 'slblock_generate' in order to generate a masked S-function block that is configured to call that S-function. The block is placed in a new model and can be copied to an existing model.

% legacy_code('slblock_generate', def);
Demoing the Generated Integration with Legacy Code

The model sldemo_lct_ndarray shows integration with the legacy code. The subsystem ndarray_add serves as a harness for the call to the legacy C function.

```matlab
open_system('sldemo_lct_ndarray')
open_system('sldemo_lct_ndarray/ndarray_add')
sim('sldemo_lct_ndarray');
```

This legacy function computes the addition of the 2 input signals:
- Input1 and Input2 are dynamically sized 3D arrays
- Output1 is a dynamically sized 3D array of same size as Input1
- the last 3 function's arguments allow to pass the Input1's dimensions to the legacy function

15-751
Complex Signals in Legacy Function

This example shows you how to use the Legacy Code Tool to integrate legacy C functions using complex signals.

The Legacy Code Tool allows you to:

• Provide the legacy function specification,
• Generate a C-MEX S-function that is used during simulation to call the legacy code, and
• Compile and build the generated S-function for simulation.

Providing the Legacy Function Specification

Functions provided with the Legacy Code Tool take a specific data structure or array of structures as the argument. The data structure is initialized by calling the function legacy_code() using 'initialize' as the first input. After initializing the structure, you have to assign its properties to values corresponding to the legacy code being integrated. For detailed help on the properties, call legacy_code('help'). The prototype of the legacy functions being called in this example is:

\[
\text{void cplx\_gain(creal\_T *input, creal\_T *gain, creal\_T *output);}
\]

where \text{creal\_T} is the complex representation of a double. The legacy source code is found in the files cplxgain.h, and cplxgain.c.

\[
% \text{sldemo\_sfun\_gain\_scalar}
\]

\[
\text{def = legacy\_code('initialize');}
\]

\[
\text{def.SFunctionName = 'sldemo\_sfun\_cplx\_gain';}
\]

\[
\text{def.OutputFcnSpec = 'void cplx\_gain(complex\<double\> u1[1], complex\<double\> p1[1], complex\<double\> y1[1]);'}
\]

\[
\text{def.HeaderFiles = {'cplxgain.h'};}\]

\[
\text{def.SourceFiles = {'cplxgain.c'};}\]

\[
\text{def.IncPaths = {'sldemo\_lct\_src'};}\]

\[
\text{def.SrcPaths = {'sldemo\_lct\_src'};}\]

Generating and Compiling an S-Function for Use During Simulation

The function legacy_code() is called again with the first input set to 'generate\_for\_sim' in order to automatically generate and compile the C-MEX S-function according to the description provided by the input argument 'def'. This S-function is used to call the legacy functions in simulation. The source code for the S-function is found in the file sldemo\_sfun\_cplx\_gain.c.
legacy_code('generate_for_sim', def);

### Start Compiling sldemo_sfun_cplx_gain
mex('-IB:\matlab\toolbox\simulink\simdemos\simfeatures\sldemo_lct_src', ... 'B:\matlab\toolbox\simulink\simdemos\simfeatures\sldemo_lct_src\cplxgain.c')
Building with 'Microsoft Visual C++ 2017 (C)'.
MEX completed successfully.
mex('sldemo_sfun_cplx_gain.c', '-IB:\matlab\toolbox\simulink\simdemos\simfeatures\sldemo_lct_src', ... 'C:\TEMP\Bdoc19b_1192687_6748\ibF7BE2B\31\tp30fbb482_509a_4051_b549_749e46c6a828\cplxgain.obj')
Building with 'Microsoft Visual C++ 2017 (C)'.
MEX completed successfully.
### Finish Compiling sldemo_sfun_cplx_gain
### Exit

Generating an rtwmakecfg.m File for Code Generation

After the TLC block file is created, the function legacy_code() can be called again with the first input set to 'rtwmakecfg_generate' in order to generate an rtwmakecfg.m file to support code generation through Simulink® Coder™. Generate the rtwmakecfg.m file if the required source and header files for the S-functions are not in the same directory as the S-functions, and you want to add these dependencies in the makefile produced during code generation.

Note: Complete this step only if you are going to simulate the model in accelerated mode.

legacy_code('rtwmakecfg_generate', def);

Generating a Masked S-Function Block for Calling the Generated S-Function

After the C-MEX S-function source is compiled, the function legacy_code() can be called again with the first input set to 'slblock_generate' in order to generate a masked S-function block that is configured to call that S-function. The block is placed in a new model and can be copied to an existing model.

% legacy_code('slblock_generate', def);

Demoing the Generated Integration with Legacy Code

The model sldemo_lct_cplxgain shows integration with the legacy code. The subsystem complex_gain serves as a harness for the call to the legacy C function via the generate S-function.

if isempty(find_system('SearchDepth',0,'Name','sldemo_lct_cplxgain'))
    open_system('sldemo_lct_cplxgain')
    open_system('sldemo_lct_cplxgain/complex_gain')
Sim('sldemo_lct_cplxgain');
end
Specified or Inherited Sample Time with Legacy Functions

This example shows you how to use the Legacy Code Tool to integrate legacy C functions with the sample time specified, inherited and parameterized.

The Legacy Code Tool allows you to:

- Provide the legacy function specification,
- Generate a C-MEX S-function that is used during simulation to call the legacy code, and
- Compile and build the generated S-function for simulation.

Providing the Legacy Function Specification

Functions provided with the Legacy Code Tool take a specific data structure or array of structures as the argument. The data structure is initialized by calling the function legacy_code() using 'initialize' as the first input. After initializing the structure, you have to assign its properties to values corresponding to the legacy code being integrated. For detailed help on the properties, call legacy_code('help'). The prototypes of the legacy functions being called in this example are:

```c
FLT gainScalar(const FLT in, const FLT gain)
```

where FLT is a typedef to float. The legacy source code is found in the files your_types.h, gain.h, and gainScalar.c.

defs = [];

% sldemo_sfun_st_inherited
def = legacy_code('initialize');
def.SFunctionName = 'sldemo_sfun_st_inherited';
def.OutputFcnSpec = 'single y1 = gainScalar(single u1, single p1)';
def.HeaderFiles = {'gain.h'};
def.SourceFiles = {'gainScalar.c'};
def.IncPaths = {'sldemo_lct_src'};
def.SrcPaths = {'sldemo_lct_src'};
defs = [defs; def];

% sldemo_sfun_st_fixed
def = legacy_code('initialize');
Generating and Compiling an S-Function for Use During Simulation

The function legacy_code() is called again with the first input set to 'generate_for_sim' in order to automatically generate and compile the C-MEX S-function according to the description provided by the input argument 'defs'. This S-function is used to call the legacy functions in simulation. The source code for the S-function is found in the file sldemo_sfun_st_inherited.c and sldemo_sfun_st_fixed.c. sldemo_sfun_st_parameterized.c.

legacy_code('generate_for_sim', defs);

### Start Compiling sldemo_sfun_st_inherited
mex('-IB:\matlab\toolbox\simulink\simdemos\simfeatures\sldemo_lct_src', ... 'B:\matlab\toolbox\simulink\simdemos\simfeatures\sldemo_lct_src\gainScalar.c')
Building with 'Microsoft Visual C++ 2017 (C)'.
MEX completed successfully.

mex('sldemo_sfun_st_inherited.c', '-IB:\matlab\toolbox\simulink\simdemos\simfeatures\sldemo_lct_src', ... 'C:\TEMP\Bdoc19b_1192687_6748\ibF7BE2B\31\tped5b2ef6_4305_4060_8440_6283b84cf63a\gainScalar.obj')
Building with 'Microsoft Visual C++ 2017 (C)'.
MEX completed successfully.

### Finish Compiling sldemo_sfun_st_inherited
### Exit

### Start Compiling sldemo_sfun_st_fixed
mex('-IB:\matlab\toolbox\simulink\simdemos\simfeatures\sldemo_lct_src', ... 'B:\matlab\toolbox\simulink\simdemos\simfeatures\sldemo_lct_src\gainScalar.c')
Building with 'Microsoft Visual C++ 2017 (C)'.
MEX completed successfully.

### Exit
Generating an rtwmakecfg.m File for Code Generation

After the TLC block file is created, the function legacy_code() can be called again with the first input set to 'rtwmakecfg_generate' in order to generate an rtwmakecfg.m file to support code generation through Simulink® Coder™. Generate the rtwmakecfg.m file if the required source and header files for the S-functions are not in the same directory as the S-functions, and you want to add these dependencies in the makefile produced during code generation.

Note: Complete this step only if you are going to simulate the model in accelerated mode.

legacy_code('rtwmakecfg_generate', def);

Generating Masked S-Function Blocks for Calling the Generated S-Functions

After the C-MEX S-function source is compiled, the function legacy_code() can be called again with the first input set to 'sldblock_generate' in order to generate masked S-function blocks which are configured to call those S-functions. The blocks are placed in a new model and can be copied to an existing model.

% legacy_code('sldblock_generate', defs);

Demoing the Generated Integration with Legacy Code

The model sldemo_lct_sampletime shows integration with the legacy code. The subsystem sample_time serves as a harness for the calls to the legacy C functions, with unit delays serving to store the previous output values.
open_system('sldemo_lct_sampletime')
open_system('sldemo_lct_sampletime/sample_time')
sim('sldemo_lct_sampletime');
System Identification for an FIR System Using MATLAB System Blocks

This example shows how to use the MATLAB System block to implement Simulink® blocks using a System object™. It highlights two MATLAB System blocks. Access the MATLAB source code for each System object by clicking the "Source Code" hyperlink from the block dialog.

- The Delay block uses a System object that has direct feedthrough set to false. This implies that the System object does not directly use the input to compute the output. Therefore, the Delay block could be safely used in a feedback system without introducing an algebraic loop.

- The LMS Adaptive Filter block estimates the coefficients of an unknown system (formed by the Unknown System and Delay blocks). Its inputs are the desired signal and the actual signal. Its outputs are the estimated signal and the vector norm of the error in the estimated coefficients.

- The Scope displays the error in the estimated signal and the vector norm of the error in the estimated coefficients.
MATLAB System Block Example:
System Identification for an FIR System using LMS Adaptive Filtering

Copyright 2013 The MathWorks, Inc.
System Identification for an FIR System Using MATLAB System Blocks
MATLAB System Block with Variable-Size Input and Output Signals

This example shows how to use the MATLAB System block to create Simulink® blocks with variable-size input and output signals.

It highlights two MATLAB System blocks. Both blocks use the same System object™ whose MATLAB source code you can access by clicking the "Source Code" hyperlink from the block dialog.

This System object inherits from the interface classes matlab.System and matlab.system.mixin.Propagates. The interface class matlab.System is required for all System objects. To use variable-size signals with a MATLAB System block, also inherit your System object from matlab.system.mixin.Propagates.

The System object specifies whether each output is fixed-size or variable-size. It also specifies the size, type, and complexity of each output. The output signals of the block represent the linear indices and elements of the input that conform to the condition specified in the block dialog.
MATLAB System Block Example:
Implement a block to find elements in the input that conform to a given condition and produces variable-size signals

Copyright 2013 The MathWorks, Inc.
Illustration of Law of Large Numbers

This example shows how to use MATLAB System blocks to illustrate the law of large numbers.

The law of large numbers states that the cumulative average probability of an event approaches the expected value as the number of trials approaches infinity. You can see this in the generated plot.

A Uniform Random Number block connected to Compare To Constant block acts as a Bernoulli trial. The example uses the high value as successful outcome of the experiment.

This example highlights the use of MATLAB System block as follows:

- The Success Counter counts the number of successful trials. The Total Trials Counter counts the total number of trials. Both blocks use the same System object™. This System object is a source block that uses the "resetImpl" method to count the number of times the block has run at consecutive time steps.

- The Analysis and Plot block calculates the average probability and plots it. It uses a System object that inherits from the interface classes matlab.System and matlab.system.mixin.Propagates. The interface class matlab.System is required for all System objects. To use features not supported for code generation, such as unsupported functions, inherit your System object from matlab.system.mixin.Propagates. As a result, this block cannot simulate in "Code generation" mode. It must simulate in "Interpreted execution" mode.

- The two MATLAB System blocks use custom icons. The Analysis and Plot block uses an image file icon. The Total Trials Counter block uses a multiline text icon. The "getIconImpl" methods of the corresponding System objects define these custom icons.
MATLAB System Block Example: Illustration of Law of Large Numbers

Copyright 2013 The Mathworks, Inc.
Using Buses with MATLAB System Blocks

This example shows how to use buses as inputs and outputs to the MATLAB System block.

- The Gain block uses a System object™ that applies the gain specified in the block to the bus elements 'a' and 'b' of the incoming bus. The output of the block is the same bus type as the input.

- The Math Operations block uses a System object that computes the sum, product, and difference between the two bus elements 'a' and 'b' of the input bus. The output of the block is a bus with three elements: 'sum', 'prod', and 'diff'.

- Note, if a System block output bus is the same type as the input bus, you must still specify the output bus name in the 'getOutputDataTypeImpl' method. This is because the 'propagatedInputDataType' method does not return the input bus object during bus propagation.

- Before model initialization, all bus types used as either inputs or outputs to the MATLAB System block must be available in the MATLAB base workspace.
Run Quality Checks on S-Functions

This example demonstrates how to use S-Function analyzer APIs to check S-functions, in order to identify potential problems and improvements.

The S-Function analyzer MATLAB command line APIs consist of:

- **Simulink.sfunction.Analyzer** - The class used to launch S-function analyzer and get analysis results.
- **Simulink.sfunction.analyzer.BuildInfo** - The class used to create an object to specify the build information for an S-function.
- **Simulink.sfunction.analyzer.Options** - The class used to specify the running options for S-function Analyzer.
- **Simulink.sfunction.analyzer.findSfunctions** - The static method to return all feasible S-functions in a model or library for S-function analyzer.

See the help for more detailed information about the above APIs.

**Specify the Input Model or Library (required)**

The S-Function analyzer requires a model or a library in MATLAB path as input, which contains the S-function blocks to be analyzed.

```matlab
model = 'slexSfunctionCheckExample';
```

**Specify the Build Information for S-functions (optional)**

S-Function analyzer checks the source code of S-functions when they are available. Heuristics are applied to automatically locate the source code based on S-function names. For example, if S-function source code and the input model are in the same folder, the source code will be included in the analysis automatically. Otherwise, the build information has to be specified through Simulink.sfunction.analyzer.BuildInfo. If no source code is available, this section can be skipped.

To specify the build information, first determine the eligible S-functions in the input model, then create a Simulink.sfunction.analyzer.BuildInfo object for each S-function. For S-function slexBadSFcn, there are two associated source files: slexBadSFcn.c and slexBadSFcn_wrapper.c

```matlab
sfunctions= Simulink.sfunction.analyzer.findSfunctions(model);
bdInfo= Simulink.sfunction.analyzer.BuildInfo(fullfile(matlabroot,'toolbox','simulink','simdemos','simfeatures','src','slexBadSFcn.c'),...
Specify the Running Options for S-function Analyzer (optional)

The execution options for S-function analyzer can be configured through Simulink.sfunction.analyzer.Options object, such as whether to enable Polyspace checks and Parameter Robustness checks, model simulation timeout, and output path for result report. If no specific options are needed, this section can be skipped. All default options will be applied.

```matlab
opts = Simulink.sfunction.analyzer.Options();
opts.EnableRobustness = 1;
```

Run the S-function Analyzer and See the Result (required)

```matlab
sfunAnalyzer = Simulink.sfunction.Analyzer(model,'BuildInfo',{bdInfo},'Options',opts);
analysisResult=sfunAnalyzer.run();
sfunAnalyzer.generateReport();
```
Interpolation Algorithm Selection and Trajectory Animation

This example shows how to use a dynamic comet plot to visualize the result of changing the interpolation and extrapolation options for a 2-D data set running in an-D Lookup Table block. Algorithm options can be employed in different situations to tradeoff accuracy, smoothness, data size, and performance.

Try opening the model and using the lookup table block's dialog to change the interpolation and extrapolation algorithms. A copy of the dialog is embedded in the right hand pane in the Model Explorer and will show up after the first simulation. You can enter and tweak values for the entire model and its blocks from the Model Explorer, including settings for model configuration parameters, Simulink blocks, Stateflow® charts, and Simulink® Coder™ settings for code generation and data management.

To make the plot, data from the simulation is logged using the non-intrusive logging features. Non-intrusive logging is annotated on the block diagram by the small blue "transmitter antennas" on certain block output ports. These logging points can be set in three ways: from the MATLAB® command line using set_param(), from the Diagram > Signals & Ports > Signal & Scope Manager, and from the signal properties dialog.
Lookup Table algorithm selection demo

- Run the model
- Use the "Rotate 3-D" button on the figure toolbar to grab, rotate, and view the surface
- Change interpolation/extrapolation in the Model Explorer
- Re-run the model to see the effects

Copyright 1990-2012 The MathWorks, Inc.
Using the Prelookup and Interpolation Blocks

The Prelookup block allows you to minimize the number of index searches performed across a set of look-up tables and also to mix clipping, extrapolation, and index search algorithms within one table calculation.

Introduction

This model uses Prelookup and Interpolation Using Prelookup blocks. Note how the Prelookup blocks perform the index search portion of the computation and the Interpolation Using Prelookup blocks perform the rest:

Open the model
These blocks can be found in the Simulink® block library in the Lookup Tables sub-library:

The prelookup block allows more efficient lookups than the regular lookup block. The regular lookup block is self-checking for breakpoint consistency.

Breakpoint consistency checking between prelookup and interpolation ensures that the breakpoint arrays in the prelookup blocks are the same length as the corresponding dimensions of the connected interpolation blocks.

In this model, the breakpoint arrays have deliberately been created with incorrect lengths to illustrate the breakpoint checking.

Update diagram to see the error message.
Optimization: PreLookups + Interpolation Using Prelookup Blocks

Use of the PreLookups block should be considered for tables with intensive index searches in breakpoint sets. The Prelookup blocks allow you to perform an index search once and reuse the result in many table lookups. In this model, 3 index search results are used in 5 blocks in a total of 10 uses. Equivalent regular lookup tables such as the example to the right would have required 10 index searches instead of 3.
Optimization: Sub-table Selection Option in the Interpolation Using Prelookup Blocks

One of the Interpolation blocks in this model is configured using a capability: trailing dimensions of an n-D table can be marked as "selection dimensions", meaning that the input for that dimension is an integer used only to make a sub-table selection, such as picking a 2-D plane from a 3-D table. The sub-table is then interpolated normally.
Interpolation of sub-tables can save tremendous amounts of computation. For every dimension eliminated from interpolation, the computation almost halves. Since an N-dimensional interpolation takes \((2^N)-1\) individual interpolation operations \((y = y_{\text{low}} + f(y_{\text{high}}-y_{\text{low}}))\), even just one dimension of selection can almost double the speed of the interpolation. Extreme example: a 5-D table with 3 dimensions of sub-table selection and 2-D interpolation: 5-D interpolation would take \(2^5-1 = 31\) interpolations, but a 2-D interpolation takes only \(2^2 - 1 = 3\).

The selection ports support vectorization to allow multiple sub-table selection / interpolations in a single block.
The sub-table or multi-table mode of operation is activated by setting a positive integer for the Number of sub-table selections dimensions parameter in the block's parameter dialog. The number you set is interpreted as the number of dimensions to select from the highest dimensions. For example, if you have a 3-D table and choose 2, that means the first dimension will be interpolated and dimensions 2 and 3 will be selected:

NOTE: you can remove highlighting from a model using the View / Remove highlighting menu item or this command:

The prelookup block allows more efficient lookups than the regular lookup block. The regular lookup block is self-checking for breakpoint consistency.

Copyright 1990-2013 The MathWorks, Inc.
set_param(mdl,'HiliteAncestors','none')

**Breakpoint and Table Data Consistency Checking**

Using Prelookup and Interpolation Using Prelookup blocks introduces a need to ensure that the length of the 'Breakpoint data' parameter in the Prelookup blocks matches the corresponding dimension of the 'Table data' parameter in the connected Interpolation Using Prelookup blocks. Simulink always checks the consistency between the two blocks when updating the block diagram. Simulink reports errors if it encounters inconsistent breakpoint and table data. For example:

```
try
    evalc(['sim(', mdl, ')']);
catch E
    disp(['Error:', E.message]);
end
```

Error:The range of values for input port k2 of block '<a href="matlab:open_and_hilite_hyperlink...corresponding table dimension of 11. Specify a different table dimension or modify the range of values for input port k2.'
Saving Memory in Prelookup and Interpolation Blocks by Using Smaller Data

This example shows how to save memory in Prelookup and Interpolation blocks. The Prelookup and Interpolation Using Prelookup blocks allow the data type storing breakpoints and table data to be set explicitly. Note that the Prelookup block allows the breakpoint data to differ from the input data type, and the Interpolation Using Prelookup block allows the table data to differ from the output data type.

In this model, the breakpoints and table are stored using single precision floating-point data type, and the calculations are performed using double precision floating-point data type. This reduces the amount of breakpoint and table data memory by half.

To see this in the generated code, open the model and build it.

All the calculations are performed using double precision floating-point, but the breakpoints and the table are stored as single precision floating-point data.
Model Advisor

This example shows the capabilities of the Model Advisor to check a model or subsystem for conditions and configuration settings that can result in inaccurate or inefficient simulation of the system represented by the model or generation of inefficient code from the model. The Model Advisor produces a report that lists all the suboptimal conditions or settings that it finds, suggesting better model configuration settings where appropriate.

The Model Advisor can be started through menus or the command line and can also be configured and used through a comprehensive API.
Copyright 1990-2017 The MathWorks, Inc.
Simulink Model Discretizer

This interactive example discretizes the Actuator Model in an aircraft Simulink® model.

If you haven't already, open the Simulink aircraft model, slexAircraftExample.slx.

To open the Model Discretizer, in the Simulink Editor, on the Apps tab, under Apps, under Control Systems, click Model Discretizer. This opens the Model Discretizer.

The Simulink model window and the Model Discretizer GUI are laid out in the screen for easy navigation.

• The model tree is shown in the left panel of the Model Discretizer GUI.
• Discretization parameters are set through the right panel of the Model Discretizer GUI.
• Discretization status is shown in the lower part of the right panel.
• By default, only continuous blocks are shown in the Model Discretizer GUI.
• In the Simulink model window, blocks in red color are continuous blocks.

You can browse through the continuous blocks one by one using the model tree view. Select the View->Next continuous block menu in the Model Discretizer GUI. This causes the Actuator Model to be highlighted.

Before discretizing the Actuator Model block, set the discretization parameters: transform method, sample time, critical frequency. Critical frequency has to be specified if the transform method 'tustin with prewarping' is selected.

After setting the discretization parameters, select the Discretize->Discretize current block menu in the Model Discretizer GUI. Or you can just click the s->z button icon in the tool bar. The discretized Actuator Model is highlighted in blue color in the model.

By default, the Actuator Model was replaced by a discrete block beneath a continuous block dialog, allowing you to continue entering the Actuator Model parameters in the s-domain. Double click the Actuator Model in the model window. Two new fields 'Sample time' and 'Method' were added to the block dialog window. These parameters are used to automatically convert the continuous parameters to equivalent discrete parameters.
Introduction to Profiling Models

The following example, with the slAccelDemoF14 model, illustrates the use of Simulink® Accelerator™ profiler feature.

Activating the Profiler

To activate the Simulink® Profiler, in the Simulink Editor, on the **Debug** tab, under **Performance Advisor**, click **Activate Profiler**. You can also issue a `set_param` command at the MATLAB® command prompt.

```matlab
modelName = 'slAccelDemoF14';
load_system(modelName)
set_param(modelName,'Profile','on')
```

Running the Simulation

While the simulation runs, the profiler is collecting data for the report. Once the simulation is complete it creates an HTML report.

```matlab
open_system(modelName)
sim(modelName);
```
Viewing the Report

The generated report consists of a summary HTML-files. Hyperlinks in the summary enable you to review the detailed timing information for each function.
**Cleaning Up**

Close the model.
bdclose(modelName)
Introduction to Accelerating Models

Simulink® Accelerator™ speeds up the execution of your model, by creating and compiling C code that takes the place of the interpretive code that Simulink uses when in Normal mode (that is, when Simulink® is not in Accelerator mode). Simulink Accelerator generates the C code from your Simulink model and invokes the MATLAB® mex function to compile and dynamically link the generated code to Simulink. This code generation and compilation process happens the first time you accelerate the model and any time the model changes are significant enough to require re-generation (for example, the addition of a block).

The following example, with the slAccelDemoF14 model, illustrates the use of Simulink Accelerator.

**Measuring the Non-accelerated Time**

Open the slAccelDemoF14 model and set the stop time to 3000 seconds. Simulate the model and time how long it takes using TIC/TOC.

```matlab
modelName = 'slAccelDemoF14';
open_system(modelName);
set_param(modelName,'stoptime','3000')
tic
sim(modelName);
toc

Elapsed time is 4.386145 seconds.
```
Turning on the Accelerator

To activate the Simulink® Accelerator, in the Simulink Editor, on the **Simulation** tab, under **Simulate**, select Accelerator. You can also issue a `set_param` command at the MATLAB® command prompt.

```
set_param(modelName,'SimulationMode','Accelerator')
```

Creating a Temporary Working Directory

Since running in accelerated mode creates some files, first move to a temporary area.

```
originalDir = pwd;
tempDir = tempname;
mkdir(tempDir)
cd(tempDir)
```
Building the Accelerated Model

Build the accelerated model and simulate it. When running in Accelerator mode, you can start the model by selecting Run on the Simulation tab or by running the command:

```
sim(modelName)
```

at the MATLAB prompt. In either case, the progress of the code generation process will be displayed in the MATLAB command window. To suppress this display wrap the sim command inside an evalc.

```
evalc('sim(modelName);');
```

Simulating the Accelerated Model

Simulate the model again. Notice the model simulation runs much faster on subsequent runs.

```
tic
sim(modelName);
toc
```

Elapsed time is 2.011878 seconds.

Changing a Tunable Parameter and Simulating the Accelerated Model

Change the value of one of the tunable parameters in the model and simulate the model again. Notice the accelerated version of the model does not have to be re-generated and so the simulation still runs more quickly in Accelerated mode than in Normal mode.

```
modelWorkspace = get_param(modelName,'ModelWorkspace');
evalin(modelWorkspace,'Mw=Mw*2;')
tic
sim(modelName);
toc
```

Elapsed time is 2.892903 seconds.

Cleaning Up

Close the model and remove the generated files.

```
bdclose(modelName)
clear([modelName,'_acc'])
```
cd(originalDir)
rmkdir(tempDir,'s')
Determining Why Simulink Accelerator Is Regenerating Code

Sometimes Simulink® regenerates the simulation target for a model at the beginning of a simulation in Accelerator Mode (TM) but it is not always clear why regeneration happens. This example shows how to use Simulink® MATLAB® commands to determine why Simulink regenerates code for Accelerator Mode simulations.

Simulink's Accelerator Mode speeds up simulation of your model by creating an executable version of the model, called a simulation target, and running this target instead of interpreting the model, as is done during Normal Mode simulation. Accelerator Mode creates the simulation target by generating C code from your model and invoking the MATLAB® mex function to compile and dynamically link the generated code to Simulink.

The process of code generation and compilation happens during the first time that you simulate your model in Accelerator Mode. Additionally, code generation may happen on subsequent simulations, especially if the model changes between simulations (for example, after addition of a block). Code generation takes times and it is often desirable to avoid it, in order to maximize the number of simulations within a given time span.

Simulink uses the model's checksum to determine if the code needs to be regenerated. This checksum is an array of four integers computed by using an md5 checksum algorithm based on attributes of the model and the blocks that it contains. Any change in the model that changes the checksum causes Simulink to regenerate the simulation target for Accelerator Mode.

Sometimes, it is not clear which model change triggered a checksum change and hence code regeneration. This example shows how to investigate why Simulink needs to regenerate code for Accelerator Mode simulation for a given model and its configuration.

Create a Temporary Working Directory

Since simulating in Accelerator Mode creates some files, first move to a temporary area.

```matlab
originalDir = pwd;
tempDir = tempname;
mkdir(tempDir)
cd(tempDir)
```
Open an Example Model

We will use a simple model, slAccelDemoWhyRebuild, throughout this example.

```matlab
model = 'slAccelDemoWhyRebuild';
open_system(model)
set_param(model,'AccelVerboseBuild','on');
```

Determining Why the Simulink Accelerator Re-builds

The first time the model runs in Accelerator mode, it generates and compiles code as expected.

```matlab
simOutput = evalc(['sim(''',model,''')']);
if ~isempty(strfind(simOutput,'Building the Accelerator target for model'))
    disp('Built Simulink Accelerator mex file')
else
    disp('Did not build Simulink Accelerator mex file')
end
```

Built Simulink Accelerator mex file

If the simulation is run again, without any changes to the model, we expect that Simulink can reuse the existing code and does not need to regenerate the code. Execute the same commands to verify.

```matlab
simOutput = evalc(['sim(''',model,''')']);
if ~isempty(strfind(simOutput,'Building the Accelerator target for model'))
```
disp('Built Simulink Accelerator mex file')
else
    disp('Did not build Simulink Accelerator mex file')
end

Did not build Simulink Accelerator mex file

Now we change some parameters in the model. We will set the following settings for the block 'Integrator': - set 'Ignore limit and reset when linearizing' to 'on' - set 'Initial Condition' to '5'

set_param([model,'/Integrator'],'IgnoreLimit','on');
set_param([model,'/Integrator'],'InitialCondition','5');

When we run the simulation again, we see that Simulink regenerates the code.

simOutput = evalc(['sim('',model,''')']);
if ~isempty(strfind(simOutput,'Building the Accelerator target for model'))
    disp('Built Simulink Accelerator mex file')
else
    disp('Did not build Simulink Accelerator mex file')
end

Built Simulink Accelerator mex file

We would like to know why.

To determine if the previously generated code is still valid for the current model configuration, Simulink compares the checksum of the model as used to generate the code to the current checksum. If they are equal, the previously generated code is still valid and Simulink Accelerator Mode reuses it for the current simulation. If the values differ, Simulink Accelerator Mode regenerates and rebuilds the code. Thus, examining the details of the checksum computation can reveal why Simulink regenerated the code.

**Get the Checksum Details**

The following command gets the model checksum computation details:

[cs1,cset1]=Simulink.BlockDiagram.getChecksum(model);

The first output is the model checksum value itself. The second output gives details of what went into the checksum computation.

Let us set the modified block parameters to their original values and get the checksum and details for that configuration
set_param([model,'/ Integrator'],'IgnoreLimit',' off');
set_param([model,'/ Integrator'],'InitialCondition',' 0');
[cs2,csdet2]= Simulink.BlockDiagram.getChecksum(model);

Comparing these two checksum values is equivalent to determining if the Simulink Accelerator will regenerate code. Note that the checksum values are different, as we expect based on the fact that Simulink Accelerator regenerates code every time it runs.

if (cs1 ~= cs2)
    disp('Checksums are different')
else
    disp('Checksums are the same')
end

Checksums are different

Now that we know that the checksums differ, the next question is why. Many things go into the checksum computation, including signal data types, some block parameter values, and block connectivity information. To understand why the checksums differ, we need to see what has changed about the items used in computing the checksum. The checksum details returned as the second argument give that information.

csdet1

csdet1 =

    struct with fields:

    ContentsChecksum: [1x1 struct]
    InterfaceChecksum: [1x1 struct]
    ContentsChecksumItems: [188x1 struct]
    InterfaceChecksumItems: [47x1 struct]

The checksum details is a structure array with four fields. Two of the fields are the component checksums of the model checksum (these are called ContentsChecksum and InterfaceChecksum) and the other two are the corresponding checksum details. These details correspond to various information which went into to the computation of the two component checksums. The model [structural] checksum is a function of ContentsChecksum and InterfaceChecksum.

First, let's determine if the difference lies in the model's contents or the model's interface...
    disp('Contents checksums are different')
else
    disp('Contents checksums are the same')
end
if (csdet1.InterfaceChecksum.Value ~= csdet2.InterfaceChecksum.Value)
    disp('Interface checksums are different')
else
    disp('Interface checksums are the same')
end

Contents checksums are different
Interface checksums are the same

Use the Details to Determine Why Checksum Changed

Now that we know the change is in the ContentsChecksum, we can look at the
ContentsChecksumItems to see what has changed.

idxForDifferences=[];
for idx = 1:length(csdet1.ContentsChecksumItems)
    if (~strcmp(csdet1.ContentsChecksumItems(idx).Handle, ...
                csdet2.ContentsChecksumItems(idx).Handle))
        idxForDifferences=[idxForDifferences,idx];
        disp(['Handles different for item ',num2str(idx)]);
    end
    if (~strcmp(csdet1.ContentsChecksumItems(idx).Identifier, ...
                csdet2.ContentsChecksumItems(idx).Identifier))
        disp(['Identifiers different for item ',num2str(idx)]);
        idxForDifferences=[idxForDifferences,idx];
    end
    if(ischar(csdet1.ContentsChecksumItems(idx).Value))
        if (~strcmp(csdet1.ContentsChecksumItems(idx).Value, ...
                    csdet2.ContentsChecksumItems(idx).Value))
            disp(['String Values different for item ',num2str(idx)]);
            idxForDifferences=[idxForDifferences,idx];
        end
    end
    end
    if(isnumeric(csdet1.ContentsChecksumItems(idx).Value))
        if (csdet1.ContentsChecksumItems(idx).Value ~= ...
                csdet2.ContentsChecksumItems(idx).Value)
            disp(['Numeric values are different for item ',num2str(idx)]);
            idxForDifferences=[idxForDifferences,idx];
        end
    end
end
String Values different for item 42

Now that we know the differences are in the items at the indices listed in idxForDifferences, we can look at those items in the two ContentsChecksumItems arrays:

```matlab
blk1 = csdet1.ContentsChecksumItems(idxForDifferences(1)).Handle
blk2 = csdet2.ContentsChecksumItems(idxForDifferences(1)).Handle
id1 = csdet1.ContentsChecksumItems(idxForDifferences(1)).Identifier
id2 = csdet2.ContentsChecksumItems(idxForDifferences(1)).Identifier
```

```matlab
blk1 =
    'slAccelDemoWhyRebuild/Integrator'

blk2 =
    'slAccelDemoWhyRebuild/Integrator'

id1 =
    'IgnoreLimit'

id2 =
    'IgnoreLimit'
```

The Handle for both items is 'slAccelDemoWhyRebuild/Integrator', which indicates the block with the changing data. The identifier for both is 'IgnoreLimit' which tells us that this was the block setting which changed, resulting in a different checksum for the model. The setting for the Initial Condition of the block does not appear in the checksum detail. Therefore, we expect that if only the setting for the Initial Condition is modified, that no rebuild will occur.
Avoid a Rebuild on Successive Simulations

Now that we found the parameter which causes the checksums to differ for this workflow, we can validate the findings by keeping that parameter constant and seeing whether rebuild occurs on successive simulations.

Let's simulate the model in Accelerator mode again. We expect it to rebuild for this simulation because we changed the parameter ('IgnoreLimit') for the checksum computation above.

```matlab
simOutput = evalc(['sim('''',model,''')']);
if ~isempty(strfind(simOutput,'Building the Accelerator target for model'))
    disp('Built Simulink Accelerator mex file')
else
    disp('Did not build Simulink Accelerator mex file')
end
```

Built Simulink Accelerator mex file

Now let's only change the Initial Condition setting and simulate again. We expect that no rebuild should happen this time.

```matlab
set_param([model,'/Integrator'],'InitialCondition','-3');
simOutput = evalc(['sim('''',model,''')']);
if ~isempty(strfind(simOutput,'Building the Accelerator target for model'))
    disp('Built Simulink Accelerator mex file')
else
    disp('Did not build Simulink Accelerator mex file')
end
```

Did not build Simulink Accelerator mex file

As expected from the checksum analysis, changing the parameter for 'Initial Condition' does not cause a regeneration of code for simulation in Accelerator mode.

Cleaning Up

Close the model and remove the generated files.

```matlab
bdclose(model)
clear([model,'_acc'])
cd(originalDir)
rmdir(tempDir,'s')
```
Parallel Simulations Using Parsim: Test-Case Sweep

This example shows how you can run multiple Simulink® simulations corresponding to different test cases in the Signal Editor block using SimulationInput objects and the parsim command. The parsim command uses Parallel Computing Toolbox™, if it is available, to run simulations in parallel, otherwise the simulations are run in serial.

Model Overview

The model sldemo_suspn_3dof shown below simulates the vehicle dynamics based on the road-suspension interaction for different road profiles. The vehicle dynamics are captured in three degrees of freedom: vertical displacement, roll, and pitch. The road profile data for the left and right tires is imported into the Signal Editor block as different test cases. The Road-Suspension Interaction subsystem calculates the suspension forces on the vehicle at the four tire locations based on the road data and the current vehicle state. In the Body Dynamics subsystem these forces and the resulting pitch and roll moments are used to determine the vehicle motion in three degrees of freedom: vertical displacement, roll, and pitch.

The suspension model is simulated using different road profiles to determine if the design meets desired performance goals. Parallel Computing Toolbox is used to speed up these multiple simulations, as illustrated below.

```matlab
mdl = 'sldemo_suspn_3dof';
isModelOpen = bdIsLoaded(mdl);
on_open_system(mdl);
```
Set up Data Required for Multiple Simulations

Determine the number of cases in the Signal Editor block using the NumberOfScenarios parameter of the Signal Editor block. The number of cases is used to determine the number of iterations to run in step 3.

```matlab
sigEditBlk = [mdl '/Road Profiles'];
numCases   = str2double(get_param(sigEditBlk,'NumberOfScenarios'));
```

Create an array of `Simulink.SimulationInput` objects to define the set of simulations to run. Each `SimulationInput` object corresponds to one simulation and will be stored as an array in a variable, `in`. The mask parameter, `ActiveScenario`, specifies the sweep value for the Signal Editor block scenario. The active scenario is set for each simulation.

```matlab
for idx = numCases:-1:1
    in(idx) = Simulink.SimulationInput(mdl);
    in(idx) = setBlockParameter(in(idx), sigEditBlk, 'ActiveScenario', idx);
end
```

Note that specifying the model parameter on the `SimulationInput` object does not apply it to the model immediately. The specified value will be applied during the simulation and reverted back to its original value, if possible, after the simulation finishes.
Run Simulations in Parallel Using Parsim

Use the `parsim` function to execute the simulations in parallel. The array of `SimulationInput` objects, `in`, created in the last step is passed into the `parsim` function as the first argument. The output from the `parsim` command is an array of `Simulink.SimulationOutput` objects which is stored in the variable `out`. Set the 'ShowProgress' option to 'on' to print a progress of the simulations on the MATLAB command window.

`out = parsim(in, 'ShowProgress', 'on');`

Each `SimulationOutput` object contains the logged signal along with the `SimulationMetadata`. When running multiple simulations using `parsim`, errors are captured so that subsequent simulations can continue to run. Any errors would show up in the `ErrorMessage` property of the `SimulationOutput` object.
Plot Results

Plot the vertical vehicle displacement from the different simulations to see how the vehicle performed to the different road profiles. The signal is logged in the SimulationOutput object in the Dataset format. Use the get method to obtain the timeseries object containing the time and signal data from each element of out.

```matlab
legend_labels = cell(1,numCases);
for i = 1:numCases
    simOut = out(i);
    ts = simOut.logssout.get('vertical_disp').Values;
    ts.plot;
    legend_labels{i} = ['Run ' num2str(i)];
    hold all
end
title('Response of a 3-DoF Suspension Model')
xlabel('Time (s)');
ylabel('Vehicle vertical displacement (m)');
legend(legend_labels,'Location','NorthEastOutside');
```
Close MATLAB Workers

Last, close the parallel pool and the model if they were not previously opened.

```matlab
if(~isModelOpen)
    close_system(mdl, 0);
end
delete(gcp('nocreate'));
```

Parallel pool using the 'local' profile is shutting down.
Parallel Simulations Using Parsim: Parameter Sweep in Normal Mode

This example shows how to run multiple simulations of a Monte Carlo study in parallel by using Parallel Computing Toolbox™. Parallel execution leverages the multiple cores of your host machine to run many simulations more quickly. These simulations could also be run in parallel on computer clusters using the MATLAB Parallel Server™. This example will work even if the Parallel Computing Toolbox™ or the MATLAB Parallel Server™ is not available, but the simulations will run in serial.

**Explore Example Model**

The model `sldemo_suspn_3dof` simulates vehicle dynamics based on the interaction between road and suspension for different road profiles. The model captures vehicle dynamics in three degrees of freedom: vertical displacement, roll, and pitch. The Signal Editor block stores measured road profile data for the left and right tires as different test groups. The Road-Suspension Interaction subsystem calculates the suspension forces on the vehicle at the four tire locations based on the road data and the current vehicle state. The Body Dynamics subsystem uses these forces and the resulting pitch and roll moments to calculate the vehicle motion in each of the three degrees of freedom.

In this Monte Carlo study, you inspect the impact of the front suspension coefficients on the vehicle dynamics. You run multiple simulations, each with a different coefficient value.

```matlab
mdl = 'sldemo_suspn_3dof';
isModelOpen = bdIsLoaded(mdl);
open_system(mdl);
```
In the model, double-click the Road-Suspension Interaction block. The mask dialog box opens. The mask parameter **Front susp. damping** sets the value of the damping coefficient, 150.

For the Body Dynamics block, find the signal that exits the Vertical disp output port. This signal represents the vertical vehicle displacement over time, which the suspension damping coefficient influences.

Right-click the signal and select **Properties**.

In the Signal Properties dialog box, the **Logging and accessibility** tab has **Log signal data** checked on, indicating that the signal is configured for logging. After the simulation finishes, you can use the specified logging name, *vertical_disp*, to identify the signal and acquire the simulation output data from the SimulationOutput object.

**Prepare Parameter Inputs**

Calculate the sweep values for the coefficient as percentages of the design value ranging from 5% to 95% in increments of 10%. Store the values in a variable, *Cf_sweep*, in the base workspace.

\[
Cf\_sweep = Cf*(0.05:0.1:0.95);
\]
Determine the number of simulations to run, which is equal to the number of sweep values. Store the number in a variable, `numSims`.

```matlab
numSims = length(Cf_sweep);
```

Use a `for` loop to:

1. Create `Simulink.SimulationInput` objects for the model. Create one object per simulation. Store the objects as an array in a variable, `in`.
2. Specify the sweep value for each simulation. Identify the target mask parameter by its underlying name, `Cf`.

```matlab
for i = numSims:-1:1
    in(i) = Simulink.SimulationInput(mdl);
    in(i) = setBlockParameter(in(i), [mdl '/Road-Suspension Interaction'], 'Cf', num2str(Cf_sweep(i)));
end
```

Note that specifying the block parameter on the `SimulationInput` object does not apply it to the model immediately. The specified value will be applied during the simulation and reverted back to its original value, if possible, after the simulation finishes.

**Run Simulations in Parallel Using Parsim**

Use the `parsim` function to execute the simulations in parallel. The array of `SimulationInput` objects, `in`, created in the last step is passed into the `parsim` function as the first argument. The output from the `parsim` command is an array of `Simulink.SimulationOutput` objects which is stored in the variable `out`. Set the 'ShowProgress' option to 'on' to print a progress of the simulations on the MATLAB command window.

```matlab
out = parsim(in, 'ShowProgress', 'on');
```

```
[13-Dec-2018 12:01:49] Checking for availability of parallel pool...
Starting parallel pool (parpool) using the 'local' profile ...
Connected to the parallel pool (number of workers: 6).
[13-Dec-2018 12:04:34] Starting Simulink on parallel workers...
[13-Dec-2018 12:04:35] Configuring simulation cache folder on parallel workers...
[13-Dec-2018 12:04:36] Loading model on parallel workers...
[13-Dec-2018 12:05:00] Running simulations...
[13-Dec-2018 12:06:38] Completed 4 of 10 simulation runs
```
Each SimulationOutput object contains the logged signal along with the SimulationMetadata. When running multiple simulations using parsim, errors are captured so that subsequent simulations can continue to run. Any errors would show up in the ErrorMessage property of the SimulationOutput object.

**Plot Results**

Plot the vertical vehicle displacement from the different simulations to see how varying the damping coefficient affects the vehicle dynamics. The signal is logged in the SimulationOutput object in the Dataset format. Use the get method to obtain the timeseries object containing the time and signal data from each element of out.

```matlab
title('Response of a 3-DoF Suspension Model')
xlabel('Time (s)');
ylabel('Vehicle vertical displacement (m)');
legend(legend_labels,'Location','NorthEastOutside');```
Close MATLAB Workers

Last, close the parallel pool and the model if they were not previously opened.

```matlab
if(~isModelOpen)
    close_system(mdl, 0);
end
delete(gcp('nocreate'));
```

Parallel pool using the 'local' profile is shutting down.
This example shows how to run multiple simulations of a Monte Carlo study in parallel by using Parallel Computing Toolbox™. Parallel execution leverages the multiple cores of your host machine to run many simulations more quickly. These simulations could also be run in parallel on computer clusters using the MATLAB Parallel Server™. This example will work even if the Parallel Computing Toolbox™ or the MATLAB Parallel Server™ is not available, but the simulations will run in serial.

Explore Example Model

The model sldemo_suspn_3dof simulates vehicle dynamics based on the interaction between road and suspension for different road profiles. The model captures vehicle dynamics in three degrees of freedom: vertical displacement, roll, and pitch. The Signal Editor block stores measured road profile data for the left and right tires as different test groups. The Road-Suspension Interaction subsystem calculates the suspension forces on the vehicle at the four tire locations based on the road data and the current vehicle state. The Body Dynamics subsystem uses these forces and the resulting pitch and roll moments to calculate the vehicle motion in each of the three degrees of freedom.

In this Monte Carlo study, the vehicle mass is varied to study its effect on the vehicle dynamics. Parallel Computing Toolbox is used to speed up these multiple simulations, as illustrated below.

mdl = 'sldemo_suspn_3dof';
isModelOpen = bdIsLoaded(mdl);
open_system(mdl);
Setup to Build the Rapid Accelerator Target

The Rapid Accelerator executable for the model is built in the SetupFcn call using the buildRapidAcceleratorTarget function from the Simulink.BlockDiagram package. The buildRapidAcceleratorTarget function returns the default run-time parameter set which is assigned to a global variable, RTP, in the SetupFcn and is used in the next step to modify the parameter values. Open sldemo_parsim_paramsweep_suspn_raccel_setup in the editor to inspect the code. Note that the build process is optimized so that if the build files are already present and compatible with the model and machine architecture then it returns early.

Set up Multiple Simulations Using SimulationInput Objects

Store the sweep values in a variable, Mb_sweep, in the base workspace.

\[ Mb\_sweep = Mb*(0.5:5:45.5); \]

Determine the number of simulations to run, which is equal to the number of sweep values. Store the number in a variable, numSims.

\[ numSims = \text{length}(Mb\_sweep); \]

Use a for loop to:

Copyright 2011-2018 The MathWorks, Inc.
1. Create Simulink.SimulationInput objects for the model. Create one object per simulation. Store the objects as an array in a variable, `in`.

2. Specify the model parameters on the SimulationInput object.

```matlab
for i = numSims:-1:1
    in(i) = Simulink.SimulationInput(mdl);
    in(i) = in(i).setModelParameter('SimulationMode', 'rapid', ...
        'RapidAcceleratorUpToDateCheck', 'off');
    in(i).PreSimFcn = @(x) sldemo_parsim_paramsweep_suspn_raccel_presim(x, Mb_sweep(i));
end
```

The SimulationInput object is used to modify the model parameters. 'SimulationMode' is set to use rapid accelerator and 'RapidAcceleratorUpToDateCheck' model parameter is set to 'off' to skip up-to-date checks since there are no structural changes made to the model between simulations and the same build files can be used. Note that specifying the model parameter on the SimulationInput object does not apply it to the model immediately. The specified value will be applied during the simulation and reverted back to its original value, if possible, after the simulation finishes. Inspect the code in the PreSimFcn, `sldemo_parsim_paramsweep_suspn_raccel_presim`. It uses `modifyTunableParameters` from the Simulink.BlockDiagram package to change the parameter corresponding to vehicle mass. The first argument to the PreSimFcn is always the SimulationInput object, and is passed into the function by Simulink®. The PreSimFunction adds another model parameter to the SimulationInput object and returns it to be used for simulation.

**Run Simulations in Parallel Using Parsim**

Use the parsim function to execute the simulations in parallel. The array of SimulationInput objects, `in`, created in the last step is passed into the `parsim` function as the first argument. The output from the `parsim` command is an array of Simulink.SimulationOutput objects which is stored in the variable `out`. Set the 'ShowProgress' option to 'on' to print a progress of the simulations on the MATLAB command window. As mentioned earlier, the SetupFcn is passed as a parameter to the parsim command to build the rapid accelerator target on the workers if required.

```matlab
out = parsim(in, 'ShowProgress', 'on', ...
    'SetupFcn', @() sldemo_parsim_paramsweep_suspn_raccel_setup(mdl));
```

```
[13-Dec-2018 12:26:56] Checking for availability of parallel pool...
Starting parallel pool (parpool) using the 'local' profile ...
Connected to the parallel pool (number of workers: 6).
[13-Dec-2018 12:28:03] Starting Simulink on parallel workers...
```
Analyzing and transferring files to the workers ... done.
[13-Dec-2018 12:28:03] Configuring simulation cache folder on parallel workers...
[13-Dec-2018 12:28:04] Running SetupFcn on parallel workers...
[13-Dec-2018 12:29:02] Loading model on parallel workers...
[13-Dec-2018 12:29:06] Running simulations...
[13-Dec-2018 12:29:17] Cleaning up parallel workers...

Each SimulationOutput object contains the logged signal along with the SimulationMetadata. When running multiple simulations using parsim, errors are captured so that subsequent simulations can continue to run. Any errors would show up in the ErrorMessage property of the SimulationOutput object.

**Plot Results**

Plot the vertical vehicle displacement from the different simulations to see how varying the vehicle mass affected the vehicle dynamics. Use the get method of the SimulationOutput object to obtain the time and signal data contained in each element of simout.

```matlab
legend_labels = cell(1,numSims);
for i = 1:numSims
    simOut = out(i);  
    ts = simOut.logsout.get('vertical_disp').Values; 
    ts.plot;  
    legend_labels{i} = ['Run ' num2str(i)]; 
    hold all
end
```

```matlab
title('Response of a 3-DoF Suspension Model')
xlabel('Time (s)');
ylabel('Vehicle vertical displacement (m)');
legend(legend_labels,'Location','NorthEastOutside');
```
Close MATLAB Workers

Last, close the parallel pool and the model if they were not previously opened.

```matlab
if(~isModelOpen)
    close_system(mdl, 0);
end
delete(gcp('nocreate'));
```

Parallel pool using the 'local' profile is shutting down.
Rapid Accelerator Simulations Using Parsim

This example shows the use of Rapid Accelerator in applications that require running parallel simulations for a range of input and parameter values.

We use the engine idle speed model which simulates the idle speed of an engine. The input of this model is the voltage of the bypass air valve and the output is the idle speed.

We run parallel simulations using `parsim` with two sets of valve voltages and by independently varying two of the three gain parameters of the transfer functions over a range of two values. The following table lists the eight simulations that will be run along with the parameter values. The external inputs, `inpSets`, are created in Step 2 below, and `gain2` and `gain3` are the variables corresponding to the two gain parameters.

<table>
<thead>
<tr>
<th>Run</th>
<th>ExternalInput</th>
<th>gain2</th>
<th>gain3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><code>inpSets(1)</code></td>
<td>25</td>
<td>20</td>
</tr>
<tr>
<td>2</td>
<td><code>inpSets(1)</code></td>
<td>25</td>
<td>30</td>
</tr>
<tr>
<td>3</td>
<td><code>inpSets(1)</code></td>
<td>35</td>
<td>20</td>
</tr>
<tr>
<td>4</td>
<td><code>inpSets(1)</code></td>
<td>35</td>
<td>30</td>
</tr>
<tr>
<td>5</td>
<td><code>inpSets(2)</code></td>
<td>25</td>
<td>20</td>
</tr>
<tr>
<td>6</td>
<td><code>inpSets(2)</code></td>
<td>25</td>
<td>30</td>
</tr>
<tr>
<td>7</td>
<td><code>inpSets(2)</code></td>
<td>35</td>
<td>20</td>
</tr>
<tr>
<td>8</td>
<td><code>inpSets(2)</code></td>
<td>35</td>
<td>30</td>
</tr>
</tbody>
</table>

It is easy to customize this example for your own application by modifying the script file used to build this example. Click the link in the top left corner of this page to edit the script file. Click the link in the top right corner to run this example from MATLAB®.

**Step 1: Preparation**

First we open the model where the simulation mode has been set to Rapid Accelerator. The default input data, and the required parameters are preloaded in the base workspace.

Open model:

```matlab
mdl = 'sldemo_raccel_engine_idle_speed';
open_system(mdl);
```
Step 2: Create Input Sets

Here we perturb the default input values vector to obtain a new input values vector.

\[
inpSets(1) = \text{timeseries}(\text{inpData}, \text{time});
\]
\[
rndPertb = 0.5 + \text{rand}(\text{length}(\text{time}), 1);
\]
\[
inpSets(2) = \text{timeseries}(\text{inpData}.*\text{rndPertb}, \text{time});
\]
\[
\text{numInpSets} = \text{length}(\text{inpSets});
\]

Step 3: Create Parameter Sets

We want to see how the idle speed changes for different values of parameters gain2 and gain3. Create an array of Simulink.SimulationInput objects to specify the different parameter values and external input for each simulation. The array of SimulationInput objects is preallocated for better performance. Also note that the external input can be directly specified on the SimulationInput object instead of using a model parameter.

\[
gain2\_vals = 25:10:35;
gain3\_vals = 20:10:30;
\]
\[
\text{num\_gain2\_vals} = \text{length}(\text{gain2\_vals});
\]
\[
\text{num\_gain3\_vals} = \text{length}(\text{gain3\_vals});
\]
\[
\text{numSims} = \text{num\_gain2\_vals}\times\text{num\_gain3\_vals}\times\text{numInpSets};
in(1:\text{numSims}) = \text{Simulink.SimulationInput(mdl)};
\]
\[
\text{id}x = 1;
\]
\[
\text{for } iG2 = 1:\text{num\_gain2\_vals}

\text{for } iG3 = 1:\text{num\_gain3\_vals}

\text{for } \text{inpSetsIdx} = 1:\text{numInpSets}

\text{in}(\text{id}x) = \text{in}(\text{id}x).\text{setModelParameter('SimulationMode', 'rapid', ...}

'RapidAcceleratorUpToDateCheck', 'off', ...}

'SaveTime', 'on', ...}

'SaveOutput', 'on');

% Use setVariable to specify a new value for a variable during % simulations
\]
\[
in(\text{id}x) = \text{in}(\text{id}x).\text{setVariable('gain2', gain2\_vals(iG2));}
\]
\[
in(\text{id}x) = \text{in}(\text{id}x).\text{setVariable('gain3', gain3\_vals(iG3));}
\]
\[
in(\text{id}x).\text{ExternalInput} = \text{inpSets(inpSetsIdx)};
\]
\[
\text{id}x = \text{id}x + 1;
\]
\[
\text{end}
\]
\[
\text{end}
\]
\[
\text{end}
\]
Note that we also use the `setModelParameter` method of the `SimulationInput` object to set model parameters to run the simulations in Rapid Accelerator mode and enable logging. The Rapid Accelerator target is built using the `SetupFcn`. The Rapid Accelerator target is built once and used by all subsequent simulations saving the time required for model compilation. Here is the code for the `SetupFcn`:

```matlab
function sldemo_parallel_rapid_accel_sims_script_setup(mdl)
    % Temporarily change the current folder on the workers to an empty
    % folder so that any existing slprj folder on the client does not
    % interfere in the build process.
    currentFolder = pwd;
    tempDir = tempname;
    mkdir(tempDir);
    cd (tempDir);
    oc = onCleanup(@() cd (currentFolder));
    Simulink.BlockDiagram.buildRapidAcceleratorTarget(mdl);
end
```

**Step 4: Execute Simulations**

Use the `parsim` function to execute the simulations in parallel. The array of `SimulationInput` objects, `in`, created in the last step is passed into the `parsim` function as the first argument. Store the simulation output data in a variable, `out`, whose value is an array of `Simulink.SimulationOutput` objects. Each `SimulationOutput` object contains the logged signal along with the `SimulationMetadata`. When running multiple simulations using `parsim`, errors are captured so that subsequent simulations can continue to run. Any errors would show up in the `ErrorMessage` property of the `SimulationOutput` object.

```matlab
out = parsim(in, 'ShowProgress', 'on', ...
    'SetupFcn', @(mdl) sldemo_parallel_rapid_accel_sims_script_setup(mdl));
```

> [16-Oct-2018 15:54:39] Checking for availability of parallel pool...
Starting parallel pool (parpool) using the 'local' profile ...
Connected to the parallel pool (number of workers: 6).
Analyzing and transferring files to the workers ...done.
[16-Oct-2018 15:55:38] Running SetupFcn on parallel workers...
[16-Oct-2018 15:56:28] Loading model on parallel workers...
[16-Oct-2018 15:56:29] Running simulations...
[16-Oct-2018 15:56:34] Completed 1 of 8 simulation runs
[16-Oct-2018 15:56:34] Completed 2 of 8 simulation runs
[16-Oct-2018 15:56:34] Completed 3 of 8 simulation runs
Step 5: Plot Results

Plot the engine idle speed with respect to time for different parameter values and inputs. The output is logged in the array format and can be accessed from the SimulationOutput object.

```matlab
for i = 1:numSims
    simOut = out(i);
    t = simOut.tout;
    y = simOut.yout;
    plot(t, y)
    hold all
end
```
Step 6: Close MATLAB Workers

delete(gcp('nocreate'))

Parallel pool using the 'local' profile is shutting down.
Multiple Simulations Workflow Tips

The goal of this demo is to provide helpful tips for running multiple simulations using parallel simulation tools. This example will work even if the Parallel Computing Toolbox™ is not available, but the simulations will run in serial. We will be using the model sldemo_suspn_3dof for this example.

```matlab
mdl = 'sldemo_suspn_3dof';
isModelOpen = bdIsLoaded(mdl);
open_system(mdl);
```

### Initializing an Array of Simulink.SimulationInput Objects

Typically you will construct an array of Simulink.SimulationInput objects in order to run multiple simulations. There are several ways to initialize the array before populating it with data.

```matlab
numSims = 5;
Cf_sweep = Cf*linspace(.05,.95, numSims);
```

**Method 1: Initialize the array before the loop**

```matlab
in(numSims) = Simulink.SimulationInput;
for idx = 1:numSims
```
% Need to populate the model name since we get any empty array by default
in(idx).ModelName = 'sldemo_suspn_3dof';
in(idx) = in(idx).setVariable('Cf', Cf_sweep(idx));
end

Method 2: Initialize the array in the loop

Note that the loop variable idx starts from the largest value so that the entire array is pre-allocated.

for idx = numSims:-1:1
    % Since we are indexing from 5 to 1, the first iteration will
    % initialize the array.
    in(idx) = Simulink.SimulationInput('sldemo_suspn_3dof');
in(idx) = in(idx).setVariable('Cf', Cf_sweep(idx));
end

Setting Model and Block Parameters

The setModelParameter and setBlockParameter methods use the same parameter-value pair syntax that the set_param API uses. This means that most values you pass in to these methods should be character arrays, not their literal value.

for idx = numSims:-1:1
    in(idx) = Simulink.SimulationInput('sldemo_suspn_3dof');

    % Incorrect
    in(idx) = in(idx).setModelParameter('StartTime', 5);

    % Correct
    in(idx) = in(idx).setModelParameter('StartTime', '3');
end

Setting Variables

The setVariable method expects that you will pass the literal value you want to assign to a variable. The idea is that this closely reflects the assignin syntax.

for idx = numSims:-1:1
    in(idx) = Simulink.SimulationInput('sldemo_suspn_3dof');

    % Incorrect, Cf is expected to be a double, not a character array
    in(idx) = in(idx).setVariable('Cf', '2500');

    % Correct, Cf is a scalar double
Diagnosing Runtime Errors

Assume that you've accidentally set up an array of Simulink.SimulationInput objects with an incorrect value.

\[
\text{Mb\_sweep = linspace(0, 1500, numSims);} \\
\text{for idx = numSims:-1:1} \\
\text{\hspace{1cm} in(idx) = Simulink.SimulationInput('sldemo_suspn_3dof');} \\
\text{\hspace{1cm} in(idx) = in(idx).setVariable('Mb', Mb\_sweep(idx));} \\
\text{\hspace{1cm} in(idx) = in(idx).setModelParameter('StopTime','1');} \\
\text{end}
\]

Simulating these will cause a runtime error

\[
\text{out = sim(in);} \\
\text{[27-Aug-2019 08:45:26] Running simulations...} \\
\text{[27-Aug-2019 08:45:28] Completed 1 of 5 simulation runs. Run 1 has errors.} \\
\text{[27-Aug-2019 08:45:30] Completed 2 of 5 simulation runs} \\
\text{[27-Aug-2019 08:45:30] Completed 3 of 5 simulation runs} \\
\text{[27-Aug-2019 08:45:31] Completed 4 of 5 simulation runs} \\
\text{[27-Aug-2019 08:45:32] Completed 5 of 5 simulation runs} \\
\text{Warning: Simulation(s) with indices listed below completed with errors. Please inspect the corresponding SimulationOutput to get more details about the error: [1]} \\
\text{Outfit dually set to 0 on the first iteration} \\
\text{in(idx) = in(idx).setVariable('Mb', Mb\_sweep(idx));} \\
\text{\hspace{1cm} End}
\]

Fortunately, you can inspect the Simulink.SimulationOutput object to see any error messages that come from a simulation.

\[
\text{out(1).ErrorMessage} \\
\text{ans =} \\
\text{'Derivative of state '1' in block 'sldemo_suspn_3dof/Body Dynamics/Vertical (Z) dyn...'} \\
\]

This works to debug issues on parallel workers too.
for idx = numSims:-1:1
    in(idx) = Simulink.SimulationInput('sldemo_suspn_3dof');

    % Accidentally set the Mass to 0 on the first iteration
    in(idx) = in(idx).setVariable('Mb', Mb_sweep(idx));

    % Shorten the stop time
    in(idx) = in(idx).setModelParameter('StopTime','1');
end

out = parsim(in);

[27-Aug-2019 08:45:33] Checking for availability of parallel pool...
Starting parallel pool (parpool) using the 'local' profile ...
Connected to the parallel pool (number of workers: 12).
[27-Aug-2019 08:48:32] Starting Simulink on parallel workers...
[27-Aug-2019 08:49:21] Configuring simulation cache folder on parallel workers...
[27-Aug-2019 08:49:26] Loading model on parallel workers...
[27-Aug-2019 08:49:55] Running simulations...
[27-Aug-2019 08:50:18] Completed 3 of 5 simulation runs
[27-Aug-2019 08:50:18] Completed 4 of 5 simulation runs
[27-Aug-2019 08:50:18] Completed 5 of 5 simulation runs
Warning: Simulation(s) with indices listed below completed with errors. Please inspect the corresponding SimulationOutput to get more details about the error:
[1]
[27-Aug-2019 08:50:18] Cleaning up parallel workers...

Inspecting the Simulink.SimulationOutput reveals a non-finite derivative error.

out(1).ErrorMessage

ans =

'Derivative of state '1' in block 'sldemo_suspn_3dof/Body Dynamics/Vertical (Z) dyn'

The applyToModel method will configure your model with the settings on the SimulationInput so you can debug the problem locally.

in(1).applyToModel;
Notice that the value of the variable Mb in the base workspace changes to 0 to reflect the value that was used in the simulation corresponding to the first SimulationInput object in.

**Close MATLAB Workers**

Last, close the parallel pool and the model if they were not previously opened.

```matlab
if(~isModelOpen)
    close_system(mdl, 0);
end
delete(gcp('nocreate'));
```

Parallel pool using the 'local' profile is shutting down.
Streamline Simulink Blockset Authoring Process with Blockset Designer

This example shows how to manage artifacts associated with a Simulink blockset with a single Blockset Designer Project.

Introduction

Simulink Blockset Designer is a Project™ based environment to author a Simulink blockset. It provides one-stop stop for the development and maintenance of Simulink blockset artifacts such as source code, TLC-files, build scripts, test suites, libraries, models, and documentations. This example demonstrates how to streamline such a Simulink blockset project.

Try the Example

Run slexBlocksetDesignerExample to open the Simulink Blockset Designer project. Then click the 'Run All' button in toolstrip to start.
Importing a Co-Simulation FMU into Simulink

This model shows how to use the FMU Import block to load an FMU file. The FMU file supports execution in co-simulation mode. Simulink supports stepping back and forth, tuning parameters in between, and saving states for the FMU Import block as long as the FMU itself supports these features.

Copyright 2017 The MathWorks, Inc.
Importing a Model Exchange FMU into Simulink

This model shows how to use the FMU Import block to load an FMU file. The FMU file supports execution in Model Exchange mode.

**Bouncing Ball System**

This FMU file models a bouncing ball system, which is an example of a hybrid dynamic system. The FMU file supports FMI 2.0 Model Exchange mode. You can change acceleration due to gravity and coefficient of restitution in the FMU block dialog. You can also tune parameters during simulation.
Importing a Model Exchange FMU into Simulink
Using Bus Signals and Structure Parameters in the FMU Import Block

This model shows how to use bus signals and structure parameters in an FMU Import block. The FMU supports execution in Model Exchange mode.

**Bus Signals and Structure Parameters Model**

This FMU file models a simple system that computes the weighted average of two input bus signals, `vehicle`, packed into an array of buses. Each of the input bus signals contains four bus elements, `wheel`. Each bus element contains two scalar signals, `position` and `speed`. The FMU accepts two structure parameters, `Weight` and `Bias`, and uses them to compute the weighted average of input signals:

\[
ev.wheel(i).position = vehicle(1).wheel(i).position \cdot Weight.v1 + Bias.v1 + vehicle(2).wheel(i).position \cdot Weight.v2 + Bias.v2
\]

\[
ev.wheel(i).speed = vehicle(1).wheel(i).speed \cdot Weight.v1 + Bias.v1 + vehicle(2).wheel(i).speed \cdot Weight.v2 + Bias.v2
\]

where

\[
i = 1 \ldots 4.
\]

The FMU file supports FMI 1.0 Model Exchange mode.

The FMU Import block supports non-virtual bus types. You must create a bus object whose structure matches the associated structure in the FMU file. When naming the bus object structures, use the variable names defined in the `modelDescription.xml` of the FMU file. For more information, see FMU documentation.
Using Bus Signals and Structure Parameters in the FMU Import Block

Copyright 2017 The MathWorks, Inc.
Co-Simulation Signal Compensation

This example shows how to use co-simulation and signal compensation for interfacing signals.

In co-simulation, components (slaves) have their own local solver. During simulation, local solvers maintain their own time by integrating from the previous step to the current step using the data exchanged between components at the previous step.

Simulink (master) serves as an integration platform and performs data exchange between slaves. Slaves do not expose their internal states to the master. The master treats slaves as discrete blocks that exchange data at discrete time intervals.

Connecting these co-simulation components does not form an algebraic loop. Instead, it introduces one-step delay during data exchange. This one-step delay can cause simulation to be less accurate or unstable.

To mitigate this issue, Simulink automatically identifies interfacing signals between these components. These signals are ideally continuous quantities that must be sampled due to co-simulation. To achieve better co-simulation stability and accuracy, Simulink performs numerical compensation on these signals. A 'gear' icon displays on the affected component to indicate this.

**Triple Mass-Spring System**

This example shows how to perform numerical compensation for three standalone mass-spring components (two implemented in C-MEX S-Function, one implemented using FMU Co-Simulation v2.0). These components are connected to form a triple mass-spring system. When you update the block diagram, the numerical compensation icons appear at the input ports.

The monolithic subsystem uses Simulink continuous blocks and is solved using a Simulink solver. It represents a pure form of the triple mass-spring system. Simulating the monolithic subsystem results in the most accurate output. Experiment toggling the numerical compensation behavior of the co-simulation components and compare the output of the co-simulation components with the monolithic subsystem output.
Multithread Co-Simulation

This example shows how to run co-simulation on multiple threads.

Simulink is an integration platform that supports co-simulation between components with local solvers or that involves simulation tools. For example, co-simulation can involve an S-function as a co-simulation gateway between Simulink and third-party tools or custom code. It can also involve an FMU in co-simulation mode imported to Simulink.

By default, Simulink configures all models to run on multiple threads with the MultiThreadCoSim parameter.

This example shows how to run multithreaded co-simulation of three components (two implemented in C-MEX S-Function, one implemented using FMU Co-Simulation v2.0). These components compute prime numbers to find the maximum prime number that is smaller than or equal to the given input.

Multithreaded co-simulation best suits models with computationally intensive and loosely coupled components. In this example, each component computes prime numbers using a brute-force search. The intensity of the computation is directly proportional to the given range, which is a block input that you can experiment with. Also, components exchange a scalar signal, which is the maximum prime number within the given range, at communication times.

Experiment toggling the MultiThreadCoSim parameter and measuring the wall clock time.
Multithread Co-Simulation

Copyright 2017 The MathWorks, Inc.
Pulse Width Modulation Using MATLAB System Block

This example shows how to use MATLAB System block to generate Pulse Width Modulation (PWM) signals based on the input period and duty cycle signals by using controllable sample time and setNumTicksUntilNextHit.

In this example, MATLAB System block registers a controllable sample time so that it can schedule the next hit while changing the output value. In this example, the MATLAB System block has two input ports and one output port. The first input port is the duty cycle signal and the second input port is the period signal. The MATLAB System block has two block parameters: the amplitude of the generated PWM signal and the resolution of the controllable sample time.

In this example, the MATLAB System block demonstrates the use of the function: *createSampleTime(obj, 'Type', 'Controllable', 'TickTime', obj.Resolution) to register a controllable sample time in getSampleTimeImpl method. The resolution must be a positive finite integer that defines the fundamental step size that the MATLAB System block can schedule the next hit for this sample time.

MATLAB System Block demonstrates the use of the method: *setNumTicksUntilNextHit(obj, numTicks) to schedule the next hit of the controllable sample time. The next hit happens after $t = t_{current} + numTicks \times resolution$. numTicks must be a positive integer. The MATLAB System block can use this method to schedule the execution of the controllable sample time in setupImpl and stepImpl.
Modeling Cyber-Physical Systems

Cyber-physical systems combine computer and physical systems to achieve design goals. Simulation of cyber-physical systems requires a combination of modeling techniques such as continuous-time, discrete-time, discrete-event, and finite state modeling. Simulink® and its companion products provide functionality to apply a wide range of modeling techniques and seamlessly integrate them in one simulation environment, which is ideal for modeling cyber-physical systems.

This example shows how continuous-time, discrete-event, and finite-state modeling techniques combine to simulate the behavior of a variable speed conveyor belt system. In SimEvents®, entities are discrete items of interest in a discrete-event simulation. Because passengers are discrete individuals, they are modeled by SimEvents® entities, created by the Entity Generator block. A Stateflow® chart models the operational modes and motor dynamics of the variable speed conveyor belt. Finally, the Entity Transport Delay block models passenger throughput as a function of conveyor belt dynamics, providing a bridge between the discrete-event and continuous-time domains.

Note: The example uses blocks from SimEvents® and Stateflow®. If you do not have a SimEvents or Stateflow license, you can open and simulate the model but only make basic changes such as modifying block parameters.
Model Structure

The model includes these key components:

- **Passengers** — Models the arrival of passengers as a Poisson process. The output is a sequence of SimEvents® entities corresponding to the passengers who step on the conveyor belt. The distribution of inter-arrival time ($\Delta t$) of a Poisson process is

  \[ P(\Delta t) = \lambda e^{-\lambda \Delta t} \]

  where $\lambda$ is the arrival rate. $\lambda$ is modeled by a MATLAB action in the model.
Entity Generator block for rush hour, normal hour, and free hour. The passenger arrival rate changes with time as:

\[
\lambda(t) = \begin{cases} 
2, & \text{mod}(t, 300) \in [0, 180), \text{ rush hour} \\
0.5, & \text{mod}(t, 300) \in [180, 240), \text{ normal hour} \\
0.1, & \text{mod}(t, 300) \in [240, 300), \text{ free hour}
\end{cases}
\]

- **Entity Transport Delay** — Holds the passengers on the conveyor belt until they arrive at the other terminal, based on the time delay calculated by the Stateflow chart.

- **Dynamics of conveyor belt** — Models the operation of a variable speed conveyor belt. See the Conveyor Belt Dynamics section for more details.

- **Dashboard** — Shows the runtime status of the conveyor belt. The color of the Mode Lamp indicates the mode of the conveyor belt.
Conveyor Belt Dynamics

Transfer Characteristics

{%
% initialize local parameters
aveMass = 75;
mu = 0.5*10*aveMass; % *mu*g*aveMass, [N]
maxPower = 5e4; % maximum power, [W]
V = 3; % normal mode velocity, [m/s]
L = 50; % length of belt, [L]
fBelt = 1e4; % friction of zero load, [N]
%

[n*aveMass >= 10] [maxPower/V-fBelt)/mu <= n]

Idle
% no load
Tl=100;
v=1/Tl*L;
P=(mu*n+fBelt)*v;
mode=0;

OnDemand
% keep speed
Ti=L/V;
v=v;
P=(mu*n+fBelt)*v;
mode=1;

Max
% keep power
v=maxPower/(mu*n+fBelt);
Ti = L/v;
P=(mu*n+fBelt)*v;
mode=2;

[n*aveMass < 10] [maxPower/V-fBelt)/mu > n]
A Stateflow® chart models the dynamics of the variable speed conveyor belt. Note in the chart that the velocity and power of the belt are plotted against a logarithmic scale of the load weight. The conveyor belt has these modes:

- **Idle** — The weight of the load is small. The belt maintains a low velocity to save energy. The Mode Lamp is gray in this mode.

- **OnDemand** — This is the normal operating mode, which maintains the optimal velocity for passenger comfort and throughput. The power will proportionally increase with the weight of the load. The Mode Lamp is green in this mode.

- **Max** — Maximum power mode. The weight of the load is too large for the conveyor belt to maintain the optimal velocity. The conveyor belt operates at the maximum possible velocity that does not exceed the maximum power. The Mode Lamp is red in this mode.

**Results**

The Scope and blocks in the DashBoard show the simulation results.

Simulation results: 1. Number of passengers versus simulation time. 2. Velocity (blue) and power (red) versus simulation time.
Three operation cycles are observed within a time span of 900. Each cycle has a period of 300, which aligns with the period of the arrival rate. The top plot shows the number of passengers on the conveyor belt over time, and the bottom plot shows the velocity and power of the conveyor belt. The velocity and power are normalized for better visualization.

The first two thirds of each period correspond to *rush hour*, and the number of passengers on the conveyor belt increases dramatically. Consequently, the conveyor belt enters into the **Max** mode quickly, which is characterized by the maximum output power with a velocity that is inversely proportional to the number of passengers. In the last third of each period, the airport is in the *normal hour* followed by the *free hour*. Therefore, the number of passengers on the conveyor belt drops and even becomes zero for some time.
The conveyor belt then operates in **OnDemand** and **Idle** modes accordingly. In **OnDemand** mode, the velocity is locked to a default value, and the power is proportional to the number of passengers. In **Idle** mode, both the velocity and power are maintained at low values to reduce energy consumption. Overall, the conveyor belt operates according to the load of the airport.
Schedule an Export-Function Model Using the Schedule Editor

This example shows how to view and edit the execution order of function-calls in an export-function model using the Schedule Editor. As in all export-function models, the desired functionality is modeled as function-call subsystems. These function-call subsystems define the partitions that are scheduled by the Schedule Editor.

With the Schedule Editor, you can easily view and edit the schedule of the function-calls. The behavior of the system depends on the execution order of these partitions. In this example, we change the execution order and observe its effects on the behavior of the system by simulating the model. To see the impact of editing the schedule on the simulation, we compare the model simulations before and after scheduling.

Create Partitions from Referenced Export-Function Model

To view and edit the schedule of the export-function model, reference the model.

```matlab
open_system('ThrottlePositionControlTop.slx');
```
ThrottlePositionControl is the referenced export-function model. By default, each function has an input port that can be used to trigger these functions. The Schedule Editor automatically handles these ports. To use the Schedule Editor, set the Schedule Rates With parameter to Schedule Editor.

```matlab
set_param('ThrottlePositionControlTop/ThrottleControl','ScheduleRatesWith','Schedule Editor');
```

**Establish a Simulation Baseline**

To observe the impact of scheduling on the model behavior, establish a baseline by simulating the model before editing the schedule. Simulate the model.

```matlab
sim('ThrottlePositionControlTop');
```

**Open the Schedule Editor**

To open the Schedule Editor, click **Schedule Editor** in the **Design** section of the **Modeling** tab. In the Schedule Editor, different components of the model are represented.
as partitions. Update the diagram to see the partitions. Partitions are the entry-points in the model. The Schedule Editor shows the execution order and data communications of these partitions. The arrows are data connections between the partitions that show the data flow. The dashed lines indicate that there is a delay because the source runs after the destination. The solid lines indicate that there is no delay as the source runs before the destination.

**Edit Partition Schedule**

The Execution Order shows the order the partitions run in at a given time step. Assume that the execution order of the partitions is in an imperfect state. In this case, to remove the delay, you want to run the ThrottleComposition ActuatorRun5ms partition after the ThrottleComposition ControllerRun5ms partition.

Drag ThrottleComposition ActuatorRun5ms after the ThrottleComposition ControllerRun5ms in the execution order. Observe that the delay between the ThrottleComposition ControllerRun5ms and the ThrottleComposition ActuatorRun5ms partitions changes to a dependency. Observe
that now there is no delay between the executions of `ThrottleComposition.ControllerRun5ms` and `ThrottleComposition.ActuatorRun5ms`.

**Schedule the Execution of Aperiodic Partitions**

The export-function model contains an unconstrained partition, `AccelerationPedalPositionSensor`. Suppose you want to schedule an unconstrained partition to simulate as if it were discrete. Schedule `AccelerationPedalPositionSensor` partition to run at [0:0.02:100] to observe its behavior at different instances of time. Click the unconstrained partition and enter `[(1:5000)*.02]` for Hit Times in the Property Inspector.
Compare the Runs in Simulation Data Inspector

Now, simulate the model with the changed schedule.

Open the Simulation Data Inspector. Select the two runs and compare. You can see how changing the schedule impacts the model behavior. For more information, see View Data with the Simulation Data Inspector.
Copyright 2018-2019 The MathWorks, Inc.